Nomadic π-Calculi: Expressing and Verifying Communication Infrastructure for Mobile Computation

Asis Unyapoth

Pembroke College
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

5 March 2001

Abstract

This thesis addresses the problem of verifying distributed infrastructure for mobile computation. In particular, we study language primitives for communication between mobile agents. They can be classified into two groups. At a low level there are *location dependent* primitives that require a programmer to know the current site of a mobile agent in order to communicate with it. At a high level there are *location independent* primitives that allow communication with a mobile agent irrespective of any migrations. Implementation of the high level requires delicate distributed infrastructure algorithms. In earlier work of Sewell, Wojciechowski and Pierce, the two levels were made precise as process calculi, allowing such algorithms to be expressed as encodings of the high level into the low level; a distributed programming language NOMADIC PICT has been built for experimenting with such encodings.

This thesis turns to semantics, giving a definition of the core language (with a type system) and proving correctness of an example infrastructure. This involves extending the standard semantics and proof techniques of process calculi to deal with the new notions of sites and agents. The techniques adopted include labelled transition semantics, operational equivalences and preorders (eg. expansion and coupled simulation), "up to" equivalences, and uniform receptiveness. We also develop two novel proof techniques for capturing the design intuitions regarding mobile agents: we consider *translocating* versions of operational equivalences that take migration into account, allowing compositional reasoning; and *temporary immobility*, which captures the intuition that while an agent is waiting for a lock somewhere in the system, it will not migrate.

The correctness proof of an example infrastructure is non-trivial. It involves analysing the possible reachable states of the encoding applied to an arbitrary high-level source program. We introduce an intermediate language for factoring out as many 'house-keeping' reduction steps as possible, and focusing on the partially-committed steps.

Declaration

Except where otherwise stated in the text, this thesis is the result of my own work and is not the outcome of work done in collaboration. This dissertation is not substantially the same as any that I have submitted or am currently submitting for a degree, diploma or any other qualification at any other university. No part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or any other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Acknowledgements

I would like to express my gratitude to my supervisor, Peter Sewell, who gave me directions in this research and helpful suggestions when I encountered hard problems. I thank him most of all for having been patient with my ignorance of the subject, as well as of the English grammar.

Prof. Robin Milner has been a great inspiration for this thesis. I attended his lectures on communicating automata and the π-calculus, and was immediately fascinated by this new way of looking at computation. This led me to an engagement in a project of implementing a concurrent application using the programming language Pict. In this project, I also had the opportunity of working with a co-author of Pict, Benjamin Pierce, as well as Peter Sewell. During this time, I learned a great deal about many research areas in the field of theoretical computer science, especially those involving process calculi and types. It was then that my interest in applying the π-calculus to real distributed systems began.

I wish to thank Pawel Wojciechowski, who took part in the design and implementation of Nomadic π-calculi, for many discussions on distributed algorithms and mobile computation. I also thank Lucian Wischik for proof-reading and for his comments.

During the time of this research, I regularly attended the meetings of the Opera Group, and, thanks to such meetings, I was able to keep in touch with the real world. Seminars organised by the Computer Laboratory, the Theory and Semantics Group also gave me chances to learn about much interesting ongoing research.

This research (and my education in England prior to that) was supported by the Royal Thai Government Students' Scholarship. I am greatly in debt of this many years of continuous and generous support, without which I would have no chance of studying in England.

More personally, I would like to thank my family in Thailand for tolerating my long absence, and my friends in Cambridge for keeping me sane. Thanks also to my house-mates for looking after my well-being (not to mention tolerating my eccentric behaviour) during the final period of my research.

Contents

Main Notations

Below are the main notations used in this thesis. The page numbers refer to the page in which the notation is defined.

Functions

$f : A \rightarrow B$	f is a partial function from A to B
$f \oplus a \mapsto b$	the function like f except that a is mapped to b
$f,\ a \mapsto b$	the function like f with $a \notin \mathsf{dom}(f)$ is mapped to b
$f + g$	the sum of functions f and g with $\mathsf{dom}(f) \cap \mathsf{dom}(g) = \emptyset$

Sets

$\mathcal{F}$	basic functions	21
$\mathcal{T}$	base types	18
$\mathcal{TV}$	type variables	18
$\mathcal{X}$	names	13
$\mathrm{n}\pi_{\mathsf{LD,LI}}$	high-level located processes	24
$\mathrm{n}\pi_{\mathsf{LD}}$	low-level located processes	24
IL	systems in the intermediate language	117

Structural Congruences

$\Gamma \equiv \Xi$	20	$P \equiv Q$	46
$LP \equiv LQ$	46	$Sys \equiv_{\Phi} Sys'$	130

Well-Formedness and Typing Judgements

$\vdash \Gamma$	for unlocated type contexts	35
$\vdash_{\mathtt{L}} \Gamma$	for located type contexts	35
$\vdash \Phi\ ok$	for valid system contexts	129
$\Gamma \vdash T$	for type	35
$\Gamma \vdash x@s$	for locating names	38
$\Gamma \vdash e \in T$	for values and expressions	37
$\Gamma \vdash ls \in \mathtt{List}\ T$	for lists	111
$\Gamma \vdash p \in T \triangleright \Delta$	for patterns	38
$\Gamma \vdash_a P$	for basic process	39
$\Gamma \vdash LP$	for located process	39
$\Phi \vdash Sys\ ok$	for systems in the intermediate language	128

Operational Semantics and Relations

$\Gamma \Vdash LP$	configuration	48
$\Gamma \Vdash LP \rightarrow \Gamma' \Vdash LQ$	reduction relation	48
$\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP$	labelled transition (basic processes)	52
$\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$	labelled transition (located processes)	53
$\dot{\sim}_\Gamma$	strong bisimulation	70
$\dot{\sim}_\Gamma^M$	translocating strong bisimulation	74
$\sim_\Gamma$	strong congruence	78
$\dot{\approx}_\Gamma$	weak bisimulation	70
$\dot{\approx}_\Gamma^M$	translocating weak bisimulation	74
$\approx_\Gamma$	weak congruence	78
$\dot{\succeq}_\Gamma^M$	translocating expansion	81
$\succeq_\Gamma$	expansion congruence	81
$\fallingdotseq_\Gamma$	coupled simulation	84

Auxiliary Definitions for the $\mathcal{C}$-Translation

Φ_{aux}	27	Φ_{aux}^-	108
Ω_D	106	Ω_{aux}	106

Miscellaneous

$\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$	deterministic reduction	101
$\mathtt{eval}(ev)$	evaluations	21
$\mathsf{match}(p, v)$	matching	43
$\mathsf{fv}(P)$	free variables of P	25
$\mathsf{fv}(LP)$	free variables of LP	25
$\mathsf{mayMove}(LP)$	potentially migrating agents in LP	76
$\mathsf{agents}(LP)$	agents in LP	99
$\mathsf{readA}(c, LP)$	agents which use c for input in LP	99
$\mathsf{writeA}(c, LP)$	agents which use c for output in LP	99
$\mathsf{fv}(T)$	free variables in T	19
$\mathsf{fv}(\Gamma)$	free variables in Γ	20
$\mathsf{dom}(\Gamma)$	domain of Γ	20
$\mathsf{range}(\Gamma)$	range of Γ	20
$\mathsf{agents}(\Gamma)$	agents declared in Γ	72
$\mathsf{mov}(\Gamma)$	mobile agents declared in Γ	72
$M_1 \uplus_\beta M_2$	'translocating union'	73
$\Gamma \oplus a \mapsto s$	updating Γ with a at site s	50
$\Gamma\beta$	updating Γ with β	69

Chapter 1

Introduction

The explosive increase of the Internet's popularity has quickly been exploited by users, programmers and business alike, as evident from the rapid growth of Internet applications and electronic commerce. The development of network applications are often hindered, however, by many problems particular to large area networks: those of efficiency, reliability, and security. Mobile computations, in which *agents* may move between machines, are considered by many to be a promising paradigm for thinking about and structuring network applications. Chess et al. in their assessment [CHK97] argued that, while mobile agents retain no particularly strong advantages over other alternatives in implementing certain functions, they provide a generalised framework for solving many existing problems. Furthermore, mobile agents also enable new, derivative network services and hence businesses.

An essential feature of mobile computation is the ability of agents to interact. This is how they access resources (on physical machines as well as on other agents) and exchange information. Existing technologies offer a variety of ways in which agents may interact — they can be classified as *location dependent* (LD) or *location independent* (LI) interactions. The former requires an agent to know the exact location of the target agent it wishes to interact with; the programmers must also ensure that the target agent does not migrate away while the message is routed to the destination. For ease of writing application using mobile agents, we need the latter form of communication which allows agents to interact without explicitly tracking each other's movement.

Location-independent communication is not supported by the standard network technology. Several programming languages which provide location independence (eg. Facile [TLK96] and the Join language [FGL$^+$96]) have some distributed infrastructure algorithms hard-coded into their implementations. It is problematic to apply this technology to wide-area networks for at least two reasons. First, it should be possible to provide different infrastructure algorithms for different applications, so that one may choose an algorithm with satisfactory range of performance matching the requirement of each application. Secondly, distributed algorithms are delicate and error-prone; the correctness of their behaviour is crucial, but difficult to verify without clear semantics or levels of abstraction. A wide-area programming language should therefore allow more flexibility by providing a low-level abstraction for distribution and network communication; location-independence (or other higher-level abstraction) can then be expressed in terms of the low-level abstraction, using the modularisation facilities of the language.

In [SWP99], the two levels of abstraction were made precise by giving them corresponding high- and low-level *Nomadic π-calculi*. The calculi are extensions of the asynchronous π-calculus [MPW92] with the notions of sites and agents. Programming using these notions requires new primitives: the low-level calculus adds those for agent creation, migration of agents between sites, and location-dependent communication between agents. To these, the high-level calculus adds a primitive for location-independent communication, suitable for writing applications. A distributed infrastructure algorithm for supporting location-independent communication can then be expressed as an encoding from the high-level calculus to the low-level calculus. The earlier work of Sewell et al. [SWP99] gave the syntax, a reduction semantics of Nomadic π-calculi, as well as two such encodings, based on central-forwarding-server and forwarding-pointer algorithms. A programming language based on the calculi, Nomadic Pict, has been implemented by Wojciechowski [WS99, Woj00a], building on the Pict implementation of Pierce and Turner [PT00].

The focus of this thesis is on developing the semantics and proof techniques for verifying distributed infrastructure algorithms. This involves extending the existing techniques of process calculi to deal with the new notions of sites and mobile agents. Mobile agents, in particular, require novel semantic techniques for capturing design intuitions, such as "while an agent is waiting for an acknowledgement from the daemon, it may not migrate." Being able to verify distributed infrastructures gives one an assurance that agents may always communicate in the location-independent mode, even in the most complex programs involving frequently-migrating agents. It is also a step towards a semantic foundation of richer wide-area distributed computing, which one may use for verifying correctness and robustness properties of programs in the presence of failure and malicious attack.

This chapter introduces some of the concepts and keywords relating to mobile computations. In Section 1.1, we give a broad overview of mobility in wide-area networks. We briefly discuss various means, advantages and areas of mobility (including process migration, mobile-device computing and mobile agents). Section 1.2 discusses the distributed infrastructure needed for supporting mobility and gives various existing examples. We give a few reasons why they are seen as problematic and a major challenge in employing mobility in wide-area networks. Section 1.3 discusses verification of such infrastructures: why it is necessary, why it is hard, and what support is required. We describe the Nomadic π-calculi in Section 1.4 and give reasons why they are suitable for such a task. In all these sections, we refrain from discussing in detail the related implementation issues. Readers may refer to various overviews and surveys [FPV98, MDW99, Woj00a] for further details and background. In Section 1.5, we outline the contribution this thesis made to research on the semantics and verification of programming languages with mobility. We conclude this chapter by giving the outline of the content of this thesis.

1.1 An Overview of Mobile Computation

The term "mobile computation" is used in several contexts, and can sometimes cause confusion. Milojičić et al. in [MDW99] discussed mobility in three major areas: process migration, mobile computation and mobile agents.

- **Process migration** is the act of transferring a process between two computers. A process here is an operating system abstraction that comprises the code, data and operating system state associated with an instance of a running application. Traditionally, process migration is used for enabling load distribution (by moving processes to lightly-loaded machines) and fault resilience (by moving processes from machines that are likely to fail). An aim of systems supporting this type of mobility is to provide transparency to the users, making processes appear as though they were running on the same machine.

  Systems supporting process migration can be classified as those which are integrated with the operating system and those which are running at the user level. The former includes the operating systems MOSIX [BL85, BS85], Sprite [OCD$^+$87, DO91], and Charlotte [AF89]. The latter are typically less efficient, but simple to maintain and to port to new systems. These include Condor [LS92] and Emerald [JLHB88].

- **Mobile computation** involves the physical movement of hardware, such as laptop and palmtop computers. These devices are becoming increasingly popular; many of today's mobile phones, for example, provide Internet access and electronic mail. Physical mobility shares similarities with "logical" process mobility, but there are specific issues as well — for example, *disconnected operation* (an ability of devices to perform computation while disconnected to the network), and the problem of limited resources (eg. battery). Readers may refer to [FZ94] for an overview of challenges in this area. This type of mobility is beyond the scope of this thesis. We shall use the terms *device mobility* when we need to refer this type of mobility. We also use the term *mobile-device network* for referring to network which supports device mobility.

- **Mobile agents** are units of executing computation that can move between machines in the network, autonomously executing tasks on behalf of users. Here Milojičić et al. distinguished mobile agents from *mobile code* (such as Java applets) by the fact that mobile agents also carry data and possibly threads of control, allowing the execution of an agent to be suspended and resumed once it moves to another site. They are generally supported at the user level by programming languages such as Telescript [Whi95], Agent Tcl [Gra96], Aglets [LO98], Voyager [Gla98], Concordia [WPW98], Sumatra [ARS97], and JoCaml [Fes98, CF99].

  Contrary to mobile agents, in the mobile code paradigm only code — and not data or threads of control — may move from one site to another. In Java [GJS97, LY97], an application can dynamically download applets from the network and execute them locally. Other languages employing this paradigm include Facile [TLP+93], TACOMA [JRS94], and M0 [Tsc94].

All these types of mobility offer many benefits, including ability to move towards a distributed resource (and hence reduce network overhead), ease of reconfiguration, increases in reliability, and (for mobile-device networks) support for disconnected operations. Mobile agents, in particular, are intended to be used for programming wide-range of applications, including electronic commerce, software distribution and updates, information retrieval, system administration, and network management.

1.2 Infrastructures and Location-Independence

In order to employ mobility in global networks and make use of their services, some underlying mechanism (which we shall refer to as *distributed infrastructure*) is required for supporting the following.

- **Mobility** This includes support for movement of processes, agents and devices, disconnected operation, and binding to local resources. Moving an agent, for example, involves suspending its execution, encoding the agent for transmission, transmitting to the new host, decoding and resuming the execution at the new host.

- **Interaction** In performing its tasks, an agent should be able to interact with its host, with other agents, and with the users. This is how an agent will access resources (where such resources can be on its hosts or on other agents), and exchange information. In a mobile-device network, an infrastructure should ensure that packets sent to a moving host do not get lost and hence eventually reach their target.

- **Heterogeneity** Users of the global network are likely to access services from different machine environments. The distributed infrastructure must therefore allows agents to be executed on machines of any type of architecture.

- **Security** This is one of the major concerns of mobile computation: how to protect agents from hostile execution environments, and how to protect execution environments from hostile agents such as viruses and worms. An infrastructure should provide some degree of protection (cf. [Nec97]).

There are numerous examples of distributed infrastructures. Mobile IP [IJ93] is a set of IP-based protocols which enables mobile machines to keep their network connections while they move in a network environment. Running an application written in a mobility-supporting programming language (eg. Telescript) requires a wide-spread runtime system (the Telescript Protocol) which enables agent transport and execution on heterogeneous machine environment. Other examples are object-based RPC systems, such as CORBA [OMG96] and DCOM [EE98]; these provide transparent access to a distributed collection of objects, hiding the true locations of objects and details of how messages are routed to their destinations.

Here we shall concentrate on support for interaction between agents. As discussed, these can be classified as location dependent and location independent. In the first, an agent a may interact with another agent b only if a knows the exact location of b. There are many examples of this: Telescript agents must be at the same place in order to interact (using an explicit *meet* operation); Agent Tcl uses an RPC-like mechanism, allowing client agents to access services of (static) server agents, whose locations can be looked up from a name-server agent. In the second, an agent can interact with another agent without knowing its location. This allows agents to communicate without explicitly tracking movements of one another. This is supported by many languages, such as Facile [TLP+93], Voyager [Gla98], the Join language [FG96, FM97], MOA [MCR+96], and Mobile Object Workbench (MOW) [BHDH98].

Location independence, however, is not supported by standard network technologies. It requires an infrastructure for tracking the movements of agents. Facile uses node servers to forward messages on a channel to the site where such a channel is created. In Voyager, an object may send a message to another object via its 'virtual reference' to object; these forward messages to the actual target remote object. MOA and MOW use an explicit name server for tracking locations of agents — agents are therefore required to register with the name server and inform it whenever they move. The Join Language also uses similar mechanism, although the interaction between agents and the name server is hidden in the implementation.

Sewell, Wojciechowski and Pierce [SWP99] argued that application of distributed infrastructures in wide-area networks is problematic for many reasons. First of all, distributed infrastructure are somewhat application-specific — different applications require different degrees of mobility, interactions and fault tolerance. This argument recalls that of Waldo et al. [WWWK94], who argue against a unified view of objects which reside in the same machines and objects which reside in different machines. Programmers, they argued, should be aware of latency, have a model of memory access, and take into account issues of concurrency and partial failure. The lack of such an awareness can lead to systems that are unreliable and incapable of scaling beyond small groups of machines. Similar polemic, although against network transparency for supporting fault tolerance, is given by Vogels et al. [VvRB98]. The second problem of distributed infrastructures, which is central to this thesis, is that they are delicate and error-prone, which can make them difficult to reason about. We discuss this in more detail in the next section.

1.3 Verifying Infrastructures

People hope to use mobile agents in a wide range of applications, including electronic commerce. Distributed infrastructures are necessary for mobile agents, and are therefore crucial to such applications. Subtle errors of infrastructure algorithms could be disastrous — financially or otherwise. It is therefore natural to demand some sort of assurance that programs will behave as they are expected. The formal proof of correctness is known as *verification*.

Verifying distributed infrastructures is difficult without clear level of abstraction or semantic definition. The descriptions of algorithms employed in Mobile IP, and of the agent tracking mechanisms mentioned above, are given informally in natural language. This can be ambiguous since natural language text cannot sufficiently describes algorithms which are highly concurrent and require delicate mechanisms for ensuring absence of race conditions, deadlock and other errors. Besides, no formal properties can be derived from such description, given the lack of formal semantics.

There are many existing methods capable of verifying distributed algorithms — most prominent are perhaps the I/O automata model [LT87, LT88] and mobile UNITY [RMP97] (an overview of these and other methods is given in Chapter 8). The problem with the existing methods, as Garland and Lynch [GL98] argue, is the lack of formal connection between verified designs and the corresponding final code. It is often feasible to prove a distributed algorithm correct, whether by hand or by using some mechanised tools, such as theorem provers. The verified algorithm, however, must be translated by hand from the pseudo-code or other mathematical constructs in which it was expressed into a real distributed programming language (eg. C++ or Java) before it can be used in a real distributed systems. This process can be difficult, time-consuming and error-prone. A programming language for distributed systems should therefore be suitable for both verification and code generation. This can be problematic, for the features which make a language suitable for verification (axiomatic style, simplicity and nondeterminism) are different from those that make it suitable for code generation (operational style, expressive power and determinism).

1.4 Nomadic π-Calculi

The Nomadic π-calculi [SWP99] have been formulated out of the need for expressing distributed infrastructures in a precise manner, for experimenting with different underlying algorithms, and for reasoning about them. The calculi offer a two-level framework: the low-level consists of a set of well-understood, location-dependent primitives for programming mobile computations — agent creation, agent migration, and communication of asynchronous messages between agents; the high-level adds a location-independent primitive, allowing agents to interact irrespective of where they are, convenient for writing applications. Distributed infrastructures can be expressed precisely as translations from the high-level calculus to the low-level calculus. The operational semantics of the calculi provide a precise understanding of the algorithms' behaviour. This supports proofs of their correctness and, ultimately, of their robustness.

The calculi are suitable for verification of distributed and mobile computations for two reasons. Firstly, their theoretical basis is supported by the fact that it is based on an asynchronous π-calculus [MPW92], which offers a clear treatment of concurrency and process communication. The theoretical basis of the π-calculus has become solid over the years, providing a variety of techniques for reasoning about process behaviour. It has been criticised for the lack of notions of locality and distribution — arguably the two most essential features in distributed systems. Nomadic π addresses this by adding the notions of sites and agents, allowing distribution and locality to be precisely described. Secondly, programs expressed in Nomadic π-calculi can be used for generating executable code, since *Nomadic Pict*, a programming language based on the calculi, has been implemented by Wojciechowski [WS99, Woj00a, Woj00b]. It builds on the Pict language of Pierce and Turner [PT00], a concurrent, though not distributed, language based on the asynchronous π-calculus. Pict supports fine-grain concurrency and the communication of asynchronous messages. To these, low- and high-level Nomadic Pict add location-dependent and location-independent primitives, corresponding to the two calculi.

In contrast to other languages which provide location-independent primitives, Nomadic Pict allows programmers to provide their own infrastructure for an application at compile time. An arbitrary infrastructure for implementing location-independent primitives can be expressed as a user-defined translation into the low-level language, which can then be deployed dynamically at runtime. The ease of expressing infrastructure algorithms encourages programmers to experiment with wide-range of infrastructures for applications with different migration and communication patterns. The language has been used for prototyping a wide range of infrastructures, from the simplest centralised-server solution to federated algorithms supporting disconnection, suited for different applications [Woj00a, WS99].

1.5 Thesis Contribution

The main contribution of this thesis is to develop semantic theories and proof techniques of Nomadic π-calculi. Despite the strong theoretical foundation and semantic techniques of its underlying formalism, verifying distributed infrastructures in Nomadic π involves a number of difficulties. Firstly, the new notions of sites and agents, together with their primitives, require some adaptation of the existing work — type systems, operational semantics and proof techniques. The adapted foundation must then be validated by proving some crucial properties (such as subject reduction and congruence results) as well as properties which are useful in proofs. The proofs of such properties are often similar to those of existing process calculi, although the new notions and constructs generally introduce some unforeseen difficulties and complication. Secondly, new proof techniques are required to capture the design intuitions regarding mobile agents. In this thesis we develop two novel techniques. *Translocating equivalences* allows behaviour of subsystems to be tested separately, provided that the testing takes into account the possibility of agents being moved by other subsystems. *Temporary immobility* captures the intuition that while an agent is waiting for a lock somewhere in the system, it may not migrate.

The techniques are illustrated by a proof that an example algorithm is correct w.r.t. coupled simulation.

1.6 Outline

This thesis consists of four parts, with the first three organised in a linear structure. The first part defines Nomadic π in three chapters:

- Chapter 2 gives the syntax, an informal description of the primitives and an example infrastructure using a central-forwarding-server algorithm;
- Chapter 3 gives the type system, whose features includes base types, tuples, polymorphism, and input/output and static/mobile subtyping for channels and agents; and
- Chapter 4 gives two operational semantics for the calculi: the reduction semantics, which captures the informal understanding of the calculi, and the labelled transition semantics, which is required for compositional reasoning. This involves extending the standard π-calculus reduction and labelled transition semantics to deal with agent mobility, location-dependent communication, and a type system. We show some basic results such as subject reduction and the correspondence between the semantics.

Some of the material in Chapter 2 has previously been published by other authors. The description of the calculi, the example infrastructure and the reduction semantics (in Chapter 4) are drawn from the works of Sewell, Wojciechowski and Pierce [SWP99, WS99]. The precise definition of the type system is new, although an informal description of the type system for Nomadic Pict was given in [WS99, Woj00a].

The second part (Chapters 5 and 6) investigates semantic and proof techniques that are used for verification of the example algorithm. These include:

- considering *translocating* versions of operational equivalences and preorders (bisimulation [MPW92] and expansion [SM92] relations) that are preserved by certain spontaneous migrations;
- proving congruence properties of some of these, to allow compositional reasoning;

- dealing with partially-committed choices, allowing a statement of the main correctness result in terms of *coupled simulation* [PS92]; and,
- identifying properties of agents that are *temporarily immobile*, waiting on a lock somewhere in the system.

The third part of the thesis, Chapter 7, gives the proof of correctness of the example infrastructure given in Chapter 2. The structure of the proof is similar to Nestmann's correctness proof of choice encodings [NP96]. This proof involves analysing the possible reachable states of the encoding applied to an arbitrary high-level source program. We introduce an intermediate language for factoring out as many 'house-keeping' reduction steps as possible, and focusing on the partially-committed steps. An overview of the main techniques and results has been published as [US01].

The last two chapters form the conclusion of this thesis. Chapter 8 discusses related work on verification of distributed algorithms and mobility. We compare the expressiveness, the design choices and the proof techniques of many existing prominent models of distributed and mobile computation. Chapter 9 summarises and discusses the achievements of this thesis and points to future work, with emphasis on the semantics of mobile computation with failures and security.

The details of the proofs of many results used in this thesis are given in the appendices.

Chapter 2

Background

The Nomadic π-calculi [SWP99] are concurrent process calculi with communication primitives. They are based on an asynchronous π-calculus [HT91, Bou92] with various ideas originated from the join calculus [FGL$^+$96] and dpi [Sew98]. The calculi inherit many properties from the asynchronous π-calculus which are inherent in real-world distributed systems, most notably concurrency and asynchronous message passing. Nomadic π-calculi add notions of *sites* and *agents*, allowing distributed and mobile computation to be precisely described.

The calculi consist of two levels. The low-level calculus only supports location-dependent primitives, which requires an agent to know the current site of the target agent it wished to communicate with. The high-level calculus supports both location-dependent and location-independent primitives, which allows agents to communicate regardless of where they are. This two-level framework allows distributed infrastructures for supporting the high-level primitive to be treated rigorously as translations between calculi.

The design of the calculi involves a delicate trade-off — many standard distributed infrastructure algorithms require non-trivial local computation within agents, yet for the theory to be tractable the calculi must be kept as simple as possible. At the level of computation, we add primitives for agent creation, agent migration and inter-agent communication to those of an asynchronous π-calculus. Other computational constructs that will be needed, eg. for finite maps, can then be regarded as lightweight syntactic sugar, as in the programming language Pict.

This chapter introduces the calculus, giving its syntax and an informal description. Section 2.1 — intended for readers who are not familiar with the π-calculus — gives background on an asynchronous π-calculus: its primitives together with their informal description, and an operational semantics. Section 2.2 then turns to Nomadic π. We begin by informally describing the calculi with an example, illustrating basic entities: channels, agents and sites. We give the definition of types, (located) type contexts and values (Section 2.2.1, Section 2.2.2 and Section 2.2.3), before giving the syntax of the two-level calculi in Section 2.2.4, and describe each of the constructs informally. Section 2.3 gives an example distributed infrastructure for supporting LI communication, based on a central-forwarding-server algorithm. This infrastructure is to be proved correct in Chapter 7, and many of the proof techniques developed in Chapter 6 are designed with this infrastructure in mind.

2.1 Asynchronous π-Calculus

The π-calculus of Milner, Parrow and Walker [MPW92, Mil93b] is a model of concurrent computation. It emerged as an elaboration and refinement of the Calculus for Communicating Systems (CCS) [Mil89] by allowing fresh channel names to be dynamically created and exchanged in communication. This makes it expressive enough to describe dynamically reconfiguring networks. The calculus has a clear treatment of concurrency and communication, which are two of the most important features of distributed systems. For this reason, it has been used as the basis for developing many techniques for programming, specification and for reasoning about distributed systems.

The π-calculus has two kinds of entities: channels and processes. A *channel* can be thought of as an abstraction of a physical communication network. It allows processes to communicate by exchanging data. A *process* here is a running program capable of multiple simultaneous activities. It may perform an internal computation or interact with its environment by inputs or outputs. The notion of π-processes is similar to that in the field of operating systems, where a process consists of an execution environment together with one or more threads. Communication occurs when one process sends a message to a channel and another (concurrent) process acquires the message by receiving from the same channel.

There are many variants of the π-calculus. Their differences range from essentially minor choices of notation and style, to important choices that are driven by the application or theory desired. In this section we describe an asynchronous, choice-free variant of the π-calculus, similar to that of Boudol [Bou92] and of Honda and Tokoro [HT91]. This description builds on that given in [Sew00].

2.1.1 Syntax

We take an infinite set $\mathcal{X}$ of names, ranged over by $a, b, c, x, y, \ldots$, and $\tau \notin \mathcal{X}$. The *process terms*, ranged over by P, Q, R, are those defined by the following grammar.

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} & \text{null} \\
 & | & P|Q & \text{parallel composition} \\
 & | & x\,\mathtt{!}\,z & \text{output } z \text{ on channel } x \\
 & | & x\mathtt{?}y \rightarrow P & \text{input from channel } x \\
 & | & \mathtt{*}x\mathtt{?}y \rightarrow P & \text{replicated input from channel } x \\
 & | & \mathtt{new}\ y\ \mathtt{in}\ P & \text{new channel name creation}
\end{array}
$$

The term $\mathbf{0}$ represents an inactive process, which cannot perform any action. The term $P|Q$ means that P and Q are concurrently active, and can also communicate. Intuitively, an asynchronous output $x\,\mathtt{!}\,z$ sends the name z on channel x. An input process $x\mathtt{?}y \rightarrow P$ waits to receive a name on x, substitutes y in P by this name after reception, and continues with P. A replicated input $\mathtt{*}x\mathtt{?}y \rightarrow P$ behaves similarly, except that it remains after the reception and so may receive another value. Placing the restriction operator $\mathtt{new}\ y\ \mathtt{in}$ before a process P ensures that y is a fresh channel in P — ie. messages sent and received on y will never be mixed with messages sent on any other channel created elsewhere, even if such a channel happens to be named y. In $x\mathtt{?}y \rightarrow P$, $\mathtt{*}x\mathtt{?}y \rightarrow P$ and $\mathtt{new}\ y\ \mathtt{in}\ P$, the name y is bound in P. We work up to alpha conversion of bound names so as to avoid clashes of names (or *capturing*). We write $\{z/y\}P$ for the process term obtained from P by replacing all free occurrences of y by z, renaming as necessary to avoid capture.

Here we exclude the constructs for *synchronous output* and a *choice operator* $+$ from the original definition of the π-calculus. A synchronous output $x\,\mathtt{!}\,z \rightarrow P$ sends the name z on channel x, and continues with P *after* z has been received by an input process. The expression $P + Q$ denotes an external choice between P and Q: either P is allowed to proceed and Q is discarded, or vice versa. The full choice is discarded here as it is not very useful for programming in our calculi. Input-only choice, which appears more useful, can be encoded in choice-free calculi (see [NP96] for details). Asynchronous calculi share many similarities with asynchronous message delivery of packet-switched networks, and so are often used as starting points for distributed calculi. We omit discussion of other choices of primitives, such as recursion, higher-order processes, join patterns and variations of concrete syntax. Overviews and discussion can be found in [Sew00, Par00].

2.1.2 Operational Semantics

The simplest form of operational semantics of primitives of the π-calculus is a *reduction relation* between process terms. We say that P *reduces* to Q, written $P \longrightarrow Q$, if P may perform a single step of computation to become Q. We shall first give some examples of reductions before giving its formal definition.

The calculus allows communication between concurrent input and output on the same channel. For example, the concurrent components of the expression $x\,!\,z \mid x\,?\,y \rightarrow P$ can communicate. The name z is being sent along channel x, so the whole expression reduces to $\mathbf{0} \mid \{z/y\}P$. The inactive processes $\mathbf{0}$ can be discarded. To illustrate the substitution $\{z/y\}$, let P be $y\,!\,w$. This process reduction can therefore be written as:

$$x\,!\,z \mid x\,?\,y \rightarrow y\,!\,w \qquad \longrightarrow \qquad z\,!\,w$$

Observe that names are first-class values in the π-calculus: they can be used for output, for input, and also be transmitted as data. In the above example, the name z, although used as datum in $x\,!\,z$, after the reduction, is used for sending the name w.

A replicated input $*x\,?\,y \rightarrow P$ behaves like an arbitrary number of parallel copies of $x\,?\,y \rightarrow P$. The replicated input can therefore be used for constructing a server which is always ready to receive further input. Below, we show a print server which prints everything it receives to the standard output.

$$\mathsf{print}\,!\,\mathrm{foo} \mid *\mathsf{print}\,?\,y \rightarrow \mathsf{stdout}\,!\,y \qquad \longrightarrow \qquad \mathsf{stdout}\,!\,\mathrm{foo} \mid *\mathsf{print}\,?\,y \rightarrow \mathsf{stdout}\,!\,y$$

Replicated inputs also allow infinite computations. As an example, the process $*\mathsf{loop}\,?\,[\,] \rightarrow \mathsf{loop}\,!\,[\,]$ responds to a signal on loop by repeating the signal, thus leading to an infinite computation.

$$\begin{array}{rcl} \mathsf{loop}\,!\,[\,] \mid *\mathsf{loop}\,?\,[\,] \rightarrow \mathsf{loop}\,!\,[\,] & \longrightarrow & \mathsf{loop}\,!\,[\,] \mid *\mathsf{loop}\,?\,[\,] \rightarrow \mathsf{loop}\,!\,[\,] \\ & \longrightarrow & \cdots \end{array}$$

Nondeterminism occurs when there are many outputs on the same channel competing for the same input, or vice versa.

$$\begin{tikzcd} & x\,!\,a \mid x\,!\,b \mid x\,?\,y \rightarrow \mathbf{0} \arrow[dl] \arrow[dr] & \\ x\,!\,a & & x\,!\,b \end{tikzcd}$$

We may use **new**-binders to generate fresh names which are different from all other names outside its scope. Such binders can be used for preventing unintended nondeterminism. For example, we may modify the above example by binding x to $x\,\mathtt{!}\,b \mid x\mathtt{?}y \to \mathbf{0}$ — the name x in $x\,\mathtt{!}\,a$ is now different from x under the binder. In this case, the race condition does not occur.

$$x\,\mathtt{!}\,a \mid \mathtt{new}\ x\ \mathtt{in}\ (x\,\mathtt{!}\,b \mid x\mathtt{?}y \to \mathbf{0}) \qquad \longrightarrow \qquad x\,\mathtt{!}\,a \mid \mathtt{new}\ x\ \mathtt{in}\ \mathbf{0}$$

Note that the term on the left above can be alpha-converted to

$$x\,\mathtt{!}\,a \mid \mathtt{new}\ x'\ \mathtt{in}\ (x'\,\mathtt{!}\,b \mid x'\mathtt{?}y \to \mathbf{0}).$$

A private name can be transmitted by output outside its original scope. The example below shows a private name z being sent along channel x outside the scope of $\mathtt{new}\ z\ \mathtt{in}$ binder, which must therefore be extended. Alpha conversion may be used to avoid capturing of other instances of z in R. This is known as *scope extrusion*.

$$\mathtt{new}\ z\ \mathtt{in}\ (x\,\mathtt{!}\,z \mid P \mid Q) \mid x\mathtt{?}y \to R \qquad \longrightarrow \qquad \mathtt{new}\ z\ \mathtt{in}\ (P \mid Q \mid \{z/y\}R)$$

If we assume further that P has no free instances of y, the the process on the right can be written as $P \mid \mathtt{new}\ z\ \mathtt{in}\ (Q \mid \{z/y\}R)$. This demonstrates the ability of the π-calculus for modelling *dynamic reconfiguration* (earlier referred to as mobility). The channel z initially serves as a data path between processes P and Q; after the reduction, however, it serves as that between Q and $\{z/y\}R$. The combination of sending channel names and scope extrusion is the essential difference between the π-calculus and earlier process calculi such as ACP, CCS and CSP.

$$\begin{array}{rcll}
P \mid \mathbf{0} & \equiv & P & \\
P \mid Q & \equiv & Q \mid P & \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \\
\mathtt{new}\ x\ \mathtt{in}\ \mathtt{new}\ y\ \mathtt{in}\ P & \equiv & \mathtt{new}\ y\ \mathtt{in}\ \mathtt{new}\ x\ \mathtt{in}\ P & \\
P \mid \mathtt{new}\ x\ \mathtt{in}\ Q & \equiv & \mathtt{new}\ x\ \mathtt{in}\ (P \mid Q) & \text{if } x \notin \mathsf{fv}(P) \\
\mathtt{new}\ x\ \mathtt{in}\ P & \equiv & P & \text{if } x \notin \mathsf{fv}(P)
\end{array}$$

Figure 2.1: Structural congruence for a simple π-calculus

The reduction semantics for our simple π-calculus can be defined in two steps. First we define a *structural congruence* (written $\equiv$). This is an equivalence relation, which formalises the intuition that we can always rearrange a reducible process so as to enable reduction. These structural rearrangement includes changing the order of parallel composition, enlarging the scope of bindings, garbage-collecting null processes and names which will no longer be used. It is the smallest equivalence relation that is a congruence and satisfies the axioms in Figure 2.1.

The second step is to define the reduction rules, capturing intuitions explained by the above examples. The reduction relation $\longrightarrow$ is the smallest binary relation over process terms satisfying the axioms in Figure 2.2.

(PI-COMM)
$$x!z \mid x?y \to P \quad\longrightarrow\quad \{z/y\}P$$

(PI-REPLIC)
$$x!z \mid *x?y \to P \quad\longrightarrow\quad \{z/y\}P \mid *x?y \to P$$

(PI-PRL)
$$\frac{P \;\longrightarrow\; Q}{P \mid R \;\longrightarrow\; Q \mid R}$$

(PI-NEW)
$$\frac{P \;\longrightarrow\; Q}{\mathbf{new}\ x\ \mathbf{in}\ P \;\longrightarrow\; \mathbf{new}\ x\ \mathbf{in}\ Q}$$

(PI-EQUIV)
$$\frac{P' \equiv P \;\longrightarrow\; Q \equiv Q'}{P' \;\longrightarrow\; Q'}$$

Figure 2.2: Reduction rules for a simple π-calculus

The reduction semantics described only defines the internal reduction of processes. An alternative style of semantics is to give a *labelled transition* relation, specifying the potential inputs and outputs of processes. This describes the interactions of processes with their environment, and therefore is more suitable for compositional reasoning. We omit a discussion of labelled transition semantics here as it is to be treated in detail in Chapter 4.

The π-calculus is sufficiently expressive to be used as the basis for a programming language. To demonstrate this, Pierce and Turner have developed the concurrent (though not distributed) language Pict [PT00] for experimenting with programming in the π-calculus. The Pict project explored the practical applicability of their theoretical work on type systems for the π-calculus [PS96, Tur96] and on the λ-calculus type systems with subtyping [PT94, HP95, PS94b]. The type system of Pict incorporates subtyping, polymorphism and a powerful type inference mechanism.

A major drawback of using the π-calculus for reasoning about realistic distributed applications is its lack of inherent notions of distribution, locality, mobility, and security. Processes in the π-calculus seem to exist in a single contiguous location, since there exists no built-in notion of distinct locations and of how locations affect execution of processes. Consequently, we cannot precisely describe distributed or mobile computations. A growing body of literature concentrates on the idea of adding discrete locations to a process calculus and considering failures of these locations. We look at this in more detail in Section 8.1.

2.2 Nomadic π-Calculi

In this section we describe the Nomadic π-calculi informally. We begin by recapitulating from [SWP99] an example program in the low-level calculus showing how an applet server can be expressed.

$$\begin{array}{l}
\texttt{*}\mathit{getApplet}\,\texttt{?}[a\ s] \rightarrow \\
\quad \texttt{create}\ b = \\
\quad\quad \texttt{migrate to}\ s \rightarrow \\
\quad\quad\quad (\langle a@s' \rangle \mathit{ack}\,\texttt{!}\,b \mid B) \\
\quad \texttt{in}\ \mathbf{0}
\end{array}$$

It can receive requests for an applet on the channel named *getApplet*; the requests contain a pair (bound to a and s) consisting of the name of the requesting agent and the name of the site for the applet to go to. When a request is received the server creates an applet agent with a new name bound to b. This agent immediately migrates to site s. It then sends an acknowledgement to the requesting agent a (which is assumed to be on site s') containing its name. In parallel, the body B of the applet commences execution.

The example illustrates the main entities of the calculi: sites, agents and channels. *Sites* should be thought of as physical machines or, more accurately, as instantiations of the Nomadic Pict runtime system on machines; each site has a unique name. Sites are ranged over by s. Nomadic π-calculi do not explicitly address questions of network failure and reconfiguration, or of security. Sites are therefore unstructured; neither network topology nor administrative domains are represented in the calculi. *Agents* are units of executing code; an agent has a unique name and a body consisting of some process; at any moment it is located at a particular site. We let a and b range over agent names. *Channels* support communication within agents, and also provide targets for inter-agent communication — an inter-agent message will be sent to a particular channel within the destination agent. Channels are ranged over by c. New agents and channels can be created dynamically. The low-level Nomadic Pict language is built above asynchronous messaging, both within and between sites. In the current implementation inter-site messages are sent on TCP connections, created on demand, but its algorithms do not depend on the message ordering that could be provided by TCP.

The inter-agent message $\langle a@s' \rangle \mathit{ack}\,\texttt{!}\,b$ is characteristic of the low-level calculus. It is location-dependent — if agent a is in fact on site s' then the message b will be delivered, to channel *ack* in a; otherwise the message will be discarded. In the implementation at most one inter-site message is sent.

2.2.1 Types

We require a type system for Nomadic π-calculi for two reasons, the first being its usual purpose: to prevent the occurrence of execution errors during the runtime of a program. Typing infrastructure algorithms requires an expressive type system, including primitive types employed in programming languages (eg. booleans, integers, strings and tuples), as well as parametric polymorphism [PS97], since an infrastructure must be able to forward messages of any type (see the `message` and `deliver` channels in Figure 2.3). The second reason is more specific: we require a type system which provides proof techniques that can be used for proving infrastructure correct. We include input/output and static/mobile subtyping, used in deriving computation steps which are functional (see Section 6.2.2) and in composing processes that are related by translocating operational relations.

For the calculi to be tractable, we aim at a type system which is small, yet sufficient for the above requirements. We take the rich type system of the Pict language [PT00] as our starting point, discarding many features which, though useful for programming (eg. recursive types and record subtyping), are likely to cause complication. We also exclude complex data structures, such as maps, which are required for expressing infrastructures, but can be expressed as an encoding (see Section 6.5).

We take an finite set $\mathcal{TV}$ of type variables of *type variables*, ranged over by X and Y, and a set $\mathcal{T}$, ranged over by B, of *base types* provided by the standard libraries of Nomadic Pict. These base types includes `Int`, and `Bool`. The *types* defined for Nomadic π, ranged over by T, are generated by the following grammar.

$$
\begin{array}{rcll}
T & ::= & B & \text{base type} \\
 & | & \texttt{Site} & \text{site} \\
 & | & \texttt{Agent}^Z & \text{agent} \\
 & | & \hat{}^I T & \text{channel type} \\
 & | & [T_1 \ldots T_n] & \text{tuple} \\
 & | & X & \text{type variable} \\
 & | & \{X\}\, T & \text{existential}
\end{array}
$$

Channels and agent types are refined by annotating them with *capabilities*. As in [PS96], channels can be used for input only `r`, output only `w`, or both `rw`; these capabilities induce a subtyping order. In addition, agents are either static `s`, or mobile `m`, as in [CGG99]. Agent and channel capabilities are ranged over by Z and I respectively. We define a relation $\leq$ on types and stipulate that a name x of type S can be used in the context where a name of type

T is required if and only if $S \leq T$. The formal definition of this relation is given in Section 3.2.

Note that we do not provide bounded quantification, which mixes subtyping with polymorphism, as in the type system of Pict. Bounded quantification allows the abstract type X of an existential type $\{X \leq S\} T$ to be specified as a subtype of S, without revealing the exact type of X. This gives rise to great expressive power at the cost of meta-theoretic complexity. We therefore discard this feature.

The type variable X in $\{X\} T$ binds in T; we work up to alpha conversion of bound type variables. The set of free type variables of a type T, written $\mathsf{fv}(T)$, is defined as follows:

$$
\begin{array}{llllll}
\mathsf{fv}(B) & \stackrel{\text{def}}{=} & \emptyset & \mathsf{fv}(X) & \stackrel{\text{def}}{=} & X \\
\mathsf{fv}(\mathtt{Agent}^Z) & \stackrel{\text{def}}{=} & \emptyset & \mathsf{fv}(\mathtt{Site}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{fv}(\hat{}^I T) & \stackrel{\text{def}}{=} & \mathsf{fv}(T) & \mathsf{fv}(\{X\} T) & \stackrel{\text{def}}{=} & \mathsf{fv}(T)/\{X\} \\
\mathsf{fv}([T_1 \ldots T_n]) & \stackrel{\text{def}}{=} & \mathsf{fv}(T_1) \cup \ldots \cup \mathsf{fv}(T_n) & & &
\end{array}
$$

2.2.2 Type contexts

We work with *located type contexts*, ranged over by $\Gamma, \Delta, \ldots$, which assign types to names, but also specify the site where each declared agent is located. The syntax is given below:

$$\Gamma \stackrel{\text{def}}{=} \bullet \mid \Gamma,\ X \mid \Gamma,\ x : \mathtt{Agent}^Z @ s \mid \Gamma,\ x : T \qquad T \neq \mathtt{Agent}^Z$$

An example of a located type context is given below.

$$s : \mathtt{Site},\ s' : \mathtt{Site},\ c : \hat{}^{\mathtt{rw}}\mathtt{Int},\ a : \mathtt{Agent}^{\mathtt{m}} @ s,\ b : \mathtt{Agent}^{\mathtt{s}} @ s'$$

The located type context declares two sites, s and s', and a channel c, which can be used for sending or receiving integers. It also declares a mobile agent a, located at s, and a static agent b, located at s'.

In Nomadic π-calculi, names of type $\mathtt{Agent}^Z$ and $\hat{}^I T$ can be dynamically created, we may refer to such names and types as being *extensible*. Correspondingly, a located type context is said to be *extensible* if it contains no type variables and all names are of extensible types. Such type contexts can be used for binding names to located processes (see later) and can be extruded by channel communication.

Located type contexts are not only useful in typechecking, but also contain site annotations, for use in the operational semantics, and consequently in the operational relations. We refer to located type contexts as *location contexts* whenever we wish to emphasise the presence of site annotations.

The domain, range and free variables of a located type context Γ, denoted $\mathsf{dom}(\Gamma)$, $\mathsf{range}(\Gamma)$, and $\mathsf{fv}(\Gamma)$, are defined as follows.

$$
\begin{array}{lcl}
\mathsf{dom}(\bullet) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{dom}(\Gamma,\ X) & \stackrel{\text{def}}{=} & \mathsf{dom}(\Gamma) \cup \{X\} \\
\mathsf{dom}(\Gamma,\ x : \mathtt{Agent}^{Z}@s) & \stackrel{\text{def}}{=} & \mathsf{dom}(\Gamma) \cup \{x\} \\
\mathsf{dom}(\Gamma,\ x : T) & \stackrel{\text{def}}{=} & \mathsf{dom}(\Gamma) \cup \{x\} \\
\mathsf{range}(\bullet) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{range}(\Gamma,\ X) & \stackrel{\text{def}}{=} & \mathsf{range}(\Gamma) \\
\mathsf{range}(\Gamma,\ x : \mathtt{Agent}^{Z}@s) & \stackrel{\text{def}}{=} & \mathsf{range}(\Gamma) \cup \{s\} \\
\mathsf{range}(\Gamma,\ x : T) & \stackrel{\text{def}}{=} & \mathsf{range}(\Gamma) \cup \mathsf{fv}(T) \\
\mathsf{fv}(\Gamma) & \stackrel{\text{def}}{=} & \mathsf{dom}(\Gamma) \cup \mathsf{range}(\Gamma)
\end{array}
$$

Although located type contexts are defined as lists, the order in which the binders appears in a list is not always important. We define $\equiv$ to be an equivalence relation between located type contexts closed under the following rules:

$$
\begin{array}{lcll}
\Gamma_1,\ X,\ Y,\ \Gamma_2 & \equiv & \Gamma_1,\ Y,\ X,\ \Gamma_2 & X \neq Y \\
\Gamma_1,\ X,\ x : \mathtt{Agent}^{Z}@s,\ \Gamma_2 & \equiv & \Gamma_1,\ x : \mathtt{Agent}^{Z}@s,\ X,\ \Gamma_2 & \\
\Gamma_1,\ X,\ x : T,\ \Gamma_2 & \equiv & \Gamma_1,\ x : T,\ X,\ \Gamma_2 & X \notin \mathsf{fv}(T) \\
\Gamma_1,\ x_1 : T_1,\ x_2 : T_2,\ \Gamma_2 & \equiv & \Gamma_1,\ x_2 : T_2,\ x_1 : T_1,\ \Gamma_2 & x_1 \neq x_2 \\
\Gamma_1,\ x_1 : \mathtt{Agent}^{Z_1}@s_1,\ x_2 : \mathtt{Agent}^{Z_2}@s_2,\ \Gamma_2 & \equiv & \Gamma_1,\ x_2 : \mathtt{Agent}^{Z_2}@s_2,\ x_1 : \mathtt{Agent}^{Z_1}@s_1,\ \Gamma_2 & \\
 & & & x_1 \neq x_2, x_1 \neq s_2, x_2 \neq s_1 \\
\Gamma_1,\ x_1 : \mathtt{Agent}^{Z}@s,\ x_2 : T,\ \Gamma_2 & \equiv & \Gamma_1,\ x_2 : T,\ x_1 : \mathtt{Agent}^{Z}@s,\ \Gamma_2 & \\
 & & & x_1 \neq x_2, x_2 \neq s
\end{array}
$$

A located type context Γ is said to *extend* Ξ (denoted $\Gamma \geq \Xi$) if there exists Ξ' such that $\Gamma \equiv \Xi, \Xi'$; in this case, we denote Ξ' by Γ/Ξ.

2.2.3 Values, Patterns and Expressions

We let t range over *constants* — that is the members of any base type B. We assume that the sets $|B|$, $\mathcal{X}$ and $\mathcal{TV}$ are disjoint from each other and from all products. Channels allow communication of first-class *values* v, which can be constants, names, tuples and existential packages. Values can be decomposed by the receiver through the use of *patterns* p. Patterns are of similar shape as values, with an addition of a wildcard pattern _, allowing matching of any value.

$$\begin{array}{rcl} v & ::= & t \mid x \mid [v_1 \ldots v_n] \mid \{T\}\, v \\ p & ::= & \_ \mid x \mid [p_1 \ldots p_n] \mid \{X\}\, p \end{array}$$

We assume that p contains no duplicated names or type variables and that X is binding in $\{X\}\, p$.

An existential package reveals the type which is bound in the existential type. For example, if c is a channel of type $\texttt{\^{}}^{\texttt{rw}}\texttt{Int}$, an existential package $\{\texttt{Int}\}\,[c\ 5]$ has the type $T = \{X\}\,[\texttt{\^{}}^{\texttt{rw}}X\ X]$; the package reveals the type variable X as $\texttt{Int}$. If d is a channel of type $\texttt{\^{}}^{\texttt{rw}}\texttt{Bool}$ then, a package $\{\texttt{Bool}\}\,[d\ \textbf{true}]$ is also of type T. A polymorphic server may use an existential pattern $\{X\}\,[x\ y]$ for decomposing such existential packages, so that eg. the value matching y can be sent along the channel matching x.

The value grammar is extended with some basic functions to give *expressions*, ranged over by ev. Basic functions, ranged over by f, include arithmetic operations and equality tests.

$$ev \ ::= \ t \mid x \mid [ev_1 \ldots ev_n] \mid \{T\}\, ev \mid f(ev_1, \ldots, ev_n)$$

The set of all basic functions, denoted $\mathcal{F}$, is intended to include most functions which are provided by Nomadic Pict libraries. For the time being, however, we restrict the basic functions to maps from tuples of base types to a single base type — except for equality where names of any type can be compared. Expressions are computed locally (in **let** processes) and, as for values, can be matched using patterns.

The evaluation relation $\texttt{eval}(ev)$ defined over expressions is given inductively as follows:

$$\begin{array}{lcl} \texttt{eval}(t) & \stackrel{\text{def}}{=} & t \\ \texttt{eval}(x) & \stackrel{\text{def}}{=} & x \\ \texttt{eval}([ev_1 \ldots ev_n]) & \stackrel{\text{def}}{=} & [\texttt{eval}(ev_1) \ldots \texttt{eval}(ev_n)] \\ \texttt{eval}(\{T\}\, ev) & \stackrel{\text{def}}{=} & \{T\}\, \texttt{eval}(ev) \\ \texttt{eval}(f(ev_1, \ldots, ev_n)) & \stackrel{\text{def}}{=} & f(\texttt{eval}(ev_1), \ldots, \texttt{eval}(ev_n)) \end{array}$$

We stipulate that all functions $f \in \mathcal{F}$ must be *total* functions (ie. if f maps tuples of type $(B_1, \ldots, B_n)$ to base type B then, for any $t_1 \in B_1, \ldots, t_n \in B_n$, $f(t_1, \ldots, t_n)$ is defined, and is of type B). This ensures that evaluation of an expression always yields a reduction (see Section 4.4).

2.2.4 Processes

We formulate two forms of processes: basic and located processes. *Basic processes* describe the threads of execution within each agent, and *located processes* describe the overall state of the computation in which many agents may be concurrently executing. In this section, we give the syntax for basic and located processes. This section is substantially a recapitulation of [SWP99].

Basic Processes The syntax of basic process for the low-level calculus is given below:

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} \ \mid\ P|Q \ \mid\ \texttt{new}\ c : \hat{\ }^I T\ \texttt{in}\ P & \multirow{2}{*}{$\Big\}\pi$-calculus primitives} \\
 & \mid & c\texttt{!}v \ \mid\ c\texttt{?}p \rightarrow P \ \mid\ \texttt{*}c\texttt{?}p \rightarrow P & \\
 & \mid & \texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ Q & \text{conditional} \\
 & \mid & \texttt{let}\ p = ev\ \texttt{in}\ P & \text{let declaration} \\
 & \mid & \texttt{create}^Z\ a = P\ \texttt{in}\ Q & \text{creation of new agent} \\
 & \mid & \texttt{migrate to}\ s \rightarrow P & \text{agent migration} \\
 & \mid & \texttt{iflocal}\ \langle a \rangle c\texttt{!}v\ \texttt{then}\ P\ \texttt{else}\ Q & \text{inter-agent communication} \\
 & \mid & \langle a \rangle c\texttt{!}v \ \mid\ \langle a@s \rangle c\texttt{!}v & \text{sugared outputs}
\end{array}
$$

The execution of the construct $\texttt{create}^Z\ b = P\ \texttt{in}\ Q$ spawns a new agent on the current site (with mobility capability Z and body P). After the creation, Q commences execution in parallel with the rest of of the body of the spawning agent. The new agent has a unique name which may be referred to both in its body and in the spawning agent. The name b is binding in P and Q. Agent can migrate to named sites — the execution of $\texttt{migrate to}\ s \rightarrow P$ as part of an agent results in the whole agent migrating to site s. After the migration, P continues in parallel with the rest of the body of the agent.

The body of an agent consists of several basic processes in parallel — essentially many threads. It uses π-calculus style interaction primitives. Execution of $\texttt{new}\ c : \hat{\ }^I T\ \texttt{in}\ P$ creates a new unique channel name c (accessible in I mode) for carrying values of type T; c is binding in P. An output $c\texttt{!}v$ (of value v on channel c) and an input $c\texttt{?}p \rightarrow P$ in the same agent may synchronise, resulting in P with the appropriate parts of the value v bound to the formal parameters in the pattern p. A replicated input $\texttt{*}c\texttt{?}p \rightarrow P$ behaves similarly except that it remains after the synchronisation, and so may receive another value. The conditional process $\texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ Q$ allows the boolean value v to be tested for its truth value and selects a continuation process accordingly. The execution of the construct $\texttt{let}\ p = ev\ \texttt{in}\ P$ evaluates expression ev and triggers P with the appropriate parts of the evaluated value $\texttt{eval}(ev)$

bound to the formal parameters in the pattern p. In $c?p \to P$, $*c?p \to P$ and $\mathtt{let}\ p = ev\ \mathtt{in}\ P$ the names in p are binding in P.

Finally, the low-level calculus includes a single primitive for interaction between agents. The execution of $\mathtt{iflocal}\ \langle a \rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ Q$ in the body of agent b has two possible outcomes. If the agent a is on the same site as agent b then the message $c!v$ will be delivered to a (where it may later interact with an input) and P will commence execution in parallel with the rest of the body of b; otherwise the message will not be delivered and Q will execute as part of b. The construct is analogous to test-and-set operations in shared memory systems—delivering the message and starting P, or discarding it and starting Q, atomically. It can greatly simplify algorithms that involve communication with agents that may migrate away at any time, yet is still implementable locally, by the runtime systems on each site. We can express two other useful constructs in the language introduced so far: $\langle a \rangle c!v$ and $\langle a@s \rangle c!v$ attempt to deliver $c!v$ to agent a, on the current site and on s, respectively. They fail silently if a is not where it is expected to be and so are usually used only where a is predictable. They can be translated into the core calculus as follows.

$$\begin{array}{rcll}
\langle a \rangle c!v & \stackrel{\text{def}}{=} & \mathtt{iflocal}\ \langle a \rangle c!v\ \mathtt{then}\ \mathbf{0}\ \mathtt{else}\ \mathbf{0} & \\
\langle a@s \rangle c!v & \stackrel{\text{def}}{=} & \mathtt{create}^{\mathtt{m}}\ b = (\mathtt{migrate\ to}\ s \to \langle a \rangle c!v)\ \mathtt{in}\ \mathbf{0} & b \notin \mathsf{fv}(a, c, v, s)
\end{array}$$

We also introduce the following abbreviations.

$$\begin{array}{rcl}
\mathtt{iflocal}\ \langle b \rangle c!v\ \mathtt{then}\ P & \stackrel{\text{def}}{=} & \mathtt{iflocal}\ \langle b \rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ \mathbf{0} \\
\mathtt{let}\ x_1 = v_1, \ldots, x_n = v_n\ \mathtt{in}\ P & \stackrel{\text{def}}{=} & \mathtt{let}\ x_1 = v_1\ \mathtt{in}\ \ldots\ \mathtt{let}\ x_n = v_n\ \mathtt{in}\ P \\
\mathtt{new}\ x_1 : T_1,\ \ldots,\ x_n : T_n\ \mathtt{in}\ P & \stackrel{\text{def}}{=} & \mathtt{new}\ x_1 : T_1\ \mathtt{in}\ \ldots\ \mathtt{new}\ x_n : T_n\ \mathtt{in}\ P
\end{array}$$

In the execution of $\mathtt{iflocal}$ a new channel name can escape the agent where it was created, later to be used for output and/or input. Synchronisation of a local output $c!v$ and an input $c?x \to P$ only occurs within an agent, however. Consider for example the process below, executing as the body of an agent a.

$$\begin{array}{l}
\mathtt{create}^{\mathtt{m}}\ b = \\
\quad c?x \to (x!3 | x?n \to \mathbf{0}) \\
\mathtt{in} \\
\quad \mathtt{new}\ d : \mathtt{\hat{}}^{\mathtt{rw}}\mathtt{Int}\ \mathtt{in} \\
\qquad \mathtt{iflocal}\ \langle b \rangle c!d\ \mathtt{then}\ \mathbf{0} \\
\qquad |\ d!7
\end{array}$$

It has a reduction for the creation of agent b, a reduction for the $\mathtt{iflocal}$ that delivers the output $c!d$ to b, and then a local synchronisation of this output with the input on c.

Agent a then has body $d\mathtt{!}7$ and agent b has body $d\mathtt{!}3|d\mathtt{?}n\rightarrow\mathbf{0}$. Only the latter output on d can synchronise with b's input $d\mathtt{?}n\rightarrow\mathbf{0}$. For each channel name there is therefore effectively a π-calculus-style channel in each agent. The channels are distinct, in that outputs and inputs can only interact if they are in the same agent. At first sight this semantics may seem counter-intuitive, but it reconciles the conflicting requirements of expressiveness and simplicity of the calculus.

High-Level Calculus The high-level calculus adds a single location-independent communication primitive $\langle a@?\rangle c\mathtt{!}v$ to the low-level calculus.

$$P \quad ::= \langle a@?\rangle c\mathtt{!}v \qquad\qquad \text{High level: LI output}$$

The intended semantics of this is that its execution will reliably deliver the message $c\mathtt{!}v$ to agent a, irrespective of the current site of a and of any migrations. The low-level communication primitives are also available for interaction with application agents whose locations are predictable.

Located Processes The syntax of located processes, ranged over by LP, of low- and high-level calculi is as follows.

$$LP ::= @_a P \;\mid\; LP|LQ \;\mid\; \mathbf{new}\ x : \mathtt{Agent}^Z\ @s\ \mathbf{in}\ LP \;\mid\; \mathbf{new}\ x : \mathtt{\char`\^}^I T\ \mathbf{in}\ LP$$

Here the body of an agent a may be split into many parts. It may, for example, be written as $@_aP_1|\ldots|@_aP_n$. Only channels and agents (and not sites) can be created dynamically; the construct $\mathbf{new}\ x : \mathtt{Agent}^Z@s\ \mathbf{in}\ LP$ declares a new agent x (binding in LP), located at site s. A new channel can be created similarly, although such a channel is not located. We define $\mathrm{n}\pi_{\mathsf{LD,LI}}$ be the set of high-level located processes defined by the above grammar. The set of low-level located processes, $\mathrm{n}\pi_{\mathsf{LD}}$, can be obtained from $\mathrm{n}\pi_{\mathsf{LD,LI}}$ by excluding process terms which contain LI primitives.

There are two extreme possibilities for annotating location information to process terms. In one, a locator is applied to the largest possible unit, with all co-located subterms gathered into a single subterm (as in [CG98, SV99]). The other extreme is where every elementary subterm is explicitly located, eg. $@_ac\mathtt{?}b\rightarrow @_bP$. This latter approach is adopted by dpi [Sew98], whose reduction semantics allows communication between inputs and outputs which are located at different locations. The former approach is too restrictive for Nomadic π-calculi, and makes labelled transitions and operational equivalences difficult to define. The latter approach is flexible, but it contains redundant location information — some of which might not be easily implementable or desirable, for example $@_a c\mathbf{?}p \rightarrow @_b P$. In our case, structural congruence rules (STR-DISTR) and (STR-N-EXTRUDE), defined in Section 4.1, are used for projecting location information down to elementary subterms if needed.

Free and Bound Variables The free variables of P and LP denoted by $\mathsf{fv}(P)$ and $\mathsf{fv}(LP)$ are defined as those names and type variables which are not bound in **`let`** patterns, input patterns or **`new`** declaration. All bound names are subjected to alpha-conversion.

2.3 Centralised Server Translation

In this section, we present an infrastructure algorithm, expressed as translation from the source language $\mathrm{n}\pi_{\mathsf{LD,LI}}$ to the target language $\mathrm{n}\pi_{\mathsf{LD}}$. This algorithm, based on the simplest algorithm from [SWP99], is a central-forwarding-server algorithm. It uses a centralised daemon for keeping a record of all existing agents. Location-independent messages are sent to the daemon, which forwards such messages to their destination. Before an agent migrate, it informs the daemon and wait for an acknowledgement; this ensures that all messages forwarded from the daemon are delivered before the agent migrates away. After the migration, the agent tells the daemon it has finished moving and continues. When a new agent is created, the new agent registers with the daemon, telling its site. The new agent, as well as its parent, wait for an acknowledgement from the daemon before they continue. Locks are used to ensure, for example, that an agent will not migrate away while a message forwarded by the daemon is on its way.

The algorithm has been chosen to illustrate the model, the use of the calculi, and, most importantly, the proof of correctness. Algorithms used in the actual mobile agent systems would have to be more delicate, taking into account efficiency as well as robustness under partial failure.

The original algorithm has been modified in the following ways to simplify the correctness proof.

- Type annotations have been added and checked with the Nomadic Pict type checker [Woj00a] (although this does not check the static/mobile subtyping).
- The algorithm is more serialised; eg. releasing `deliver` at last moment, so that the newly-created agent is more deterministic.

- Fresh channels are used for transmitting acknowledgements, making such channels linear [KPT96]. This simplifies the proof of correctness, since communication along a linear channel yields an expansion.

- The translation is extended to arbitrary located processes (not just source programs containing a single agent). Proving operational relations involves using co-inductive proof techniques. This means that, instead of defining the translation for programs containing a single agent, we need to strengthen it so that any program can be translated.

The daemon is itself implemented as a static agent. The translation $\mathcal{C}_\Phi [\![LP]\!]$ of a located process $LP = \mathbf{new}\ \Delta\ \mathbf{in}\ (@_{a_1} P_1 \mid \ldots \mid @_{a_n} P_n)$ (well-typed with respect to Φ) then consists roughly of the daemon agent in parallel with a compositional translation $[\![P_i]\!]_{a_i}$ of each source agent; or more precisely:

$$\begin{array}{rcl} \mathcal{C}_\Phi [\![LP]\!] & \stackrel{\text{def}}{=} & \mathbf{new}\ \Delta,\ \Phi_{aux},\ m : \mathtt{Map[Agent^s\ Site]}\ \mathbf{in} \\ & & \quad @_D(\mathit{Daemon} \mid \mathtt{lock!}m \mid \mathsf{makeMap}(m; \mathsf{Enlist}(\Phi, \Delta))) \\ & & \mid \prod_{i \in \{1 \ldots n\}} @_{a_i}([\![P_i]\!]_{a_i} \mid \mathtt{currentloc!}s_i \mid \mathtt{Deliverer}) \end{array} \qquad (\dagger)$$

where each agent a_i is distinct and assumed to be located at s_i and $\mathsf{Enlist}(\Phi, \Delta)$ initialises a *site map*: a finite map from agent names to site names, recording the current site of every agent $a_1 \ldots a_n$ in the system — both free and bound. It is defined recursively below.

$$\begin{array}{rcll} \mathsf{Enlist}(\bullet) & \stackrel{\text{def}}{=} & \mathbf{nil} & \\ \mathsf{Enlist}(x : \mathtt{Agent}^Z @ s,\ \Theta) & \stackrel{\text{def}}{=} & [x\ z] \mathbin{\mathtt{::}} \mathsf{Enlist}(\Theta) & \\ \mathsf{Enlist}(x : T,\ \Theta) & \stackrel{\text{def}}{=} & \mathsf{Enlist}(\Theta) & \quad T \neq \mathtt{Agent}^Z \end{array}$$

The $\mathsf{makeMap}(m; ls)$ function, defined in Section 6.5, can then be used for generating a basic process representing the site map, accessible via m.

The body of the daemon and the compositional translation are shown in Figures 2.3 and 2.4. They interact using channels of an *interface context* Φ_{aux}, also defined in Figure 2.3. The interface context additionally declares lock channels and the daemon name D, located at a fixed site SD. The daemon uses a map type constructor, which (together with the map operators) can be translated into the core language (see Section 6.5). The process definitions on the right in Figures 2.3 and 2.4 (such as mesgQ, regReq, and regBlockC) are used extensively for defining various constructs used for the correctness proof in Chapter 7.

$$
\begin{array}{ll}
\mathit{Daemon} \stackrel{\mathrm{def}}{=} & \\
\quad \mathtt{*message?}\ \{X\}\ [a\ c\ v] \rightarrow & \\
\quad\quad \mathtt{lock?}m \rightarrow & (\mathsf{mesgReq}(\{X\}\ [a\ c\ v])) \\
\quad\quad\quad \mathbf{lookup}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ a\ \mathbf{in}\ m\ \mathbf{with} & \\
\quad\quad\quad\quad \mathbf{found}(s) \rightarrow \mathbf{new}\ \mathtt{dack} : \mathtt{\hat{}}^{\mathtt{rw}}[]\ \mathbf{in} & \\
\quad\quad\quad\quad\quad \langle a@s \rangle \mathtt{deliver!}\ \{X\}\ [c\ v\ \mathtt{dack}] & \\
\quad\quad\quad\quad\quad \mid \mathtt{dack?}[] \rightarrow \mathtt{lock!}m & \\
\quad\quad\quad\quad \mathbf{notfound} \rightarrow \mathbf{0} & \\
\quad \mid \mathtt{*register?}[b\ s\ \mathtt{rack}] \rightarrow & \\
\quad\quad \mathtt{lock?}m \rightarrow & (\mathsf{regReq}(b\ s\ \mathtt{rack})) \\
\quad\quad\quad \mathbf{let}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ m' = (m\ \mathbf{with}\ b \mapsto s)\ \mathbf{in} & \\
\quad\quad\quad\quad (\mathtt{lock!}m' \mid \langle b@s \rangle \mathtt{rack!}[]) & \\
\quad \mid \mathtt{*migrating?}[a\ \mathtt{mack}] \rightarrow & \\
\quad\quad \mathtt{lock?}m \rightarrow & (\mathsf{migReq}(a\ \mathtt{mack})) \\
\quad\quad\quad \mathbf{lookup}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ a\ \mathbf{in}\ m\ \mathbf{with} & \\
\quad\quad\quad\quad \mathbf{found}(s) \rightarrow \mathbf{new}\ \mathtt{migrated} : \mathtt{\hat{}}^{\mathtt{rw}}[\mathtt{Site}\ \mathtt{\hat{}}^{\mathtt{w}}[]]\ \mathbf{in} & \\
\quad\quad\quad\quad\quad \langle a@s \rangle \mathtt{mack!}[\mathtt{migrated}] & \\
\quad\quad\quad\quad\quad \mid \mathtt{migrated?}[s'\ \mathtt{ack}] \rightarrow & (\mathsf{migProc}(a\ m\ \mathtt{migrated})) \\
\quad\quad\quad\quad\quad\quad \mathbf{let}\ m' = (m\ \mathbf{with}\ a \mapsto s')\ \mathbf{in} & \\
\quad\quad\quad\quad\quad\quad\quad (\mathtt{lock!}m' \mid \langle a@s' \rangle \mathtt{ack!}[]) & \\
\quad\quad\quad\quad \mathbf{notfound} \rightarrow \mathbf{0} &
\end{array}
$$

$$
\begin{array}{lll}
\Phi_{aux} & \stackrel{\mathrm{def}}{=} & D : \mathtt{Agent}^{\mathtt{s}}@SD, \\
& & \mathtt{lock} : \mathtt{\hat{}}^{\mathtt{rw}}\mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}], \\
& & \mathtt{register} : \mathtt{\hat{}}^{\mathtt{rw}}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}\ \mathtt{\hat{}}^{\mathtt{w}}[]], \\
& & \mathtt{migrating} : \mathtt{\hat{}}^{\mathtt{rw}}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{\hat{}}^{\mathtt{w}}[\mathtt{\hat{}}^{\mathtt{w}}[\mathtt{Site}\ \mathtt{\hat{}}^{\mathtt{w}}[]]]], \\
& & \mathtt{message} : \mathtt{\hat{}}^{\mathtt{rw}}\ \{X\}\ [\mathtt{Agent}^{\mathtt{s}}\ \mathtt{\hat{}}^{\mathtt{w}}X\ X], \\
& & \mathtt{deliver} : \mathtt{\hat{}}^{\mathtt{rw}}\ \{X\}\ [\mathtt{\hat{}}^{\mathtt{w}}X\ X\ \mathtt{\hat{}}^{\mathtt{w}}[]], \\
& & \mathtt{currentloc} : \mathtt{\hat{}}^{\mathtt{rw}}\mathtt{Site}
\end{array}
$$

Figure 2.3: The Central Server Daemon and the Interface Context

$$[\![\langle b@?\rangle c\mathtt{!}v]\!]_a \stackrel{\text{def}}{=} \langle D@SD\rangle\mathtt{message!}\,\{T\}\,[b\ c\ v]$$

$$
\begin{array}{l}
[\![\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q]\!]_a \stackrel{\text{def}}{=} \\
\quad \mathtt{currentloc?}s \rightarrow \mathbf{new}\ \mathtt{pack} : \hat{}^{\mathtt{rw}}[],\ \mathtt{rack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \\
\quad\quad \mathbf{create}^Z\ b = \\
\quad\quad\quad \langle D@SD\rangle\mathtt{register!}[b\ s\ \mathtt{rack}] \\
\quad\quad\quad \mid \mathtt{rack?}[] \rightarrow \mathbf{iflocal}\ \langle a\rangle\mathtt{pack!}[]\ \mathbf{then} \qquad (\mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P)) \\
\quad\quad\quad\quad\quad (\mathtt{currentloc!}s \mid [\![P]\!]_b \mid \mathtt{Deliverer}) \\
\quad\quad \mathbf{in} \\
\quad\quad\quad \mathtt{pack?}[] \rightarrow (\mathtt{currentloc!}s \mid [\![Q]\!]_a) \qquad (\mathsf{regBlockP}(s\ \mathtt{pack}\ Q)) \\
\\
\quad\quad \text{where}\quad \mathtt{Deliverer} \stackrel{\text{def}}{=} \mathtt{*deliver?}\,\{X\}\,[c\ v\ \mathtt{dack}] \rightarrow (\langle D@SD\rangle\mathtt{dack!}[] \mid c\mathtt{!}v)
\end{array}
$$

$$
\begin{array}{l}
[\![\mathbf{migrate\ to}\ s \rightarrow P]\!]_a \stackrel{\text{def}}{=} \\
\quad \mathtt{currentloc?\_} \rightarrow \mathbf{new}\ \mathtt{mack} : \hat{}^{\mathtt{rw}}[\hat{}^{\mathtt{w}}[\mathtt{Site}\ \hat{}^{\mathtt{w}}[]]]\ \mathbf{in} \\
\quad\quad \langle D@SD\rangle\mathtt{migrating!}[a\ \mathtt{mack}] \\
\quad\quad \mid \mathtt{mack?}[\mathtt{migrated}] \rightarrow \qquad (\mathsf{migBlock}(s\ \mathtt{mack}\ P)) \\
\quad\quad\quad \mathbf{migrate\ to}\ s \rightarrow \mathbf{new}\ \mathtt{ack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \qquad (\mathsf{migReady}(s\ \mathtt{migrated}\ P)) \\
\quad\quad\quad\quad (\langle D@SD\rangle\mathtt{migrated!}[s\ \mathtt{ack}] \\
\quad\quad\quad\quad \mid \mathtt{ack?}[] \rightarrow \mathtt{currentloc!}s \mid [\![P]\!]_a)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\mathbf{0}]\!]_a & \stackrel{\text{def}}{=} & \mathbf{0} \\
[\![P|Q]\!]_a & \stackrel{\text{def}}{=} & [\![P]\!]_a \mid [\![Q]\!]_a \\
[\![c\mathtt{?}p \rightarrow P]\!]_a & \stackrel{\text{def}}{=} & c\mathtt{?}p \rightarrow [\![P]\!]_a \\
[\![\mathtt{*}c\mathtt{?}p \rightarrow P]\!]_a & \stackrel{\text{def}}{=} & \mathtt{*}c\mathtt{?}p \rightarrow [\![P]\!]_a \\
[\![c\mathtt{!}v]\!]_a & \stackrel{\text{def}}{=} & c\mathtt{!}v \\
[\![\mathbf{iflocal}\ \langle b\rangle c\mathtt{!}v\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!]_a & \stackrel{\text{def}}{=} & \mathbf{iflocal}\ \langle b\rangle c\mathtt{!}v\ \mathbf{then}\ [\![P]\!]_a\ \mathbf{then}\ [\![Q]\!]_a \\
[\![\mathbf{new}\ x : \hat{}^{I}T\ \mathbf{in}\ P]\!]_a & \stackrel{\text{def}}{=} & \mathbf{new}\ x : \hat{}^{I}T\ \mathbf{in}\ [\![P]\!]_a \\
[\![\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!]_a & \stackrel{\text{def}}{=} & \mathbf{if}\ v\ \mathbf{then}\ [\![P]\!]_a\ \mathbf{then}\ [\![Q]\!]_a \\
[\![\mathbf{let}\ p = ev\ \mathbf{in}\ P]\!]_a & \stackrel{\text{def}}{=} & \mathbf{let}\ p = ev\ \mathbf{in}\ [\![P]\!]_a
\end{array}
$$

Figure 2.4: The Compositional Encoding

The daemon consists of three replicated inputs, on the `message`, `register`, and `migrating` channels, ready to receive messages from the encodings of agents. Reacting to such messages *Daemon* produces one of the following *request* processes:

- a registration request, $\mathsf{regReq}(b\ s\ \mathtt{ack})$: an agent b requests to be registered with an initial site s;
- a migrating request, $\mathsf{migReq}(a\ \mathtt{ack})$: an agent a requests permission to migrate; and
- a message forwarding request, $\mathsf{mesgReq}(\{T\}\ [a\ c\ v])$: an agent requests the daemon to forward a message $c\,\mathtt{!}\,v$ to agent a.

In making registration and migrating requests, the requesting agent needs to supply a fresh channel `ack` which the daemon can use for sending acknowledgement once the request has been processed.

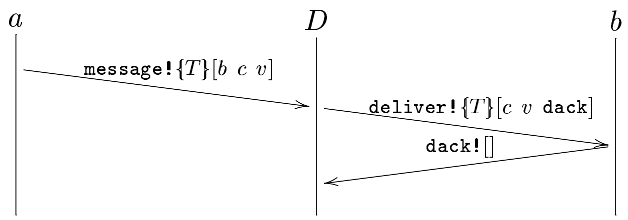
Part of the initialisation code places *Daemon* in parallel with an output on `lock` which carries a reference to the site map of the daemon. This ensures that the daemon is essentially single-threaded, since each of the above requests begins with an input on `lock` (thereby acquiring both the lock and the site map), and does not relinquish the lock until the daemon finishes with the request. The code preserves the invariant that at any time there is at most one output on `lock`.

Each agent records its current site internally as an output on its `currentloc`. This channel is also used as a lock, to enforce mutual exclusion between the encodings of all agent creation and migration commands within the body of the agent. The local lock is not essential for the correctness of the algorithm but it makes the proof of correctness simpler, since the lock ensures that at anytime an agent can be involved with at most one request to the daemon. This simplifies the intermediate language (see Section 7.2), as well as working with partially committed agents. If an agents could involve with more than one daemon requests, committing its partially committed actions (as in Section 7.4) can be problematic: at which site would an agent involved with several migration requests be placed when committed, for example.

Turning to the compositional translation $[\![.]\!]$, it is defined inductively on the syntax of basic processes, allowing properties of $[\![.]\!]$ to be proved by induction. Only three clauses, shown in Figure 2.4, are non-trivial: for the location-independent output, agent creation and agent migration primitives. For the rest, $[\![.]\!]$ is homomorphic. We assume in the definition that none of the names in `typewriter` font (which, by convention, are auxiliary names defined in Φ_{aux}) occur in x (for the definition of $[\![\mathbf{new}\ x : \hat{\ }^I T\ \mathbf{in}\ P]\!]_a$) and in p (for the definitions of processes containing pattern p).

Location-Independent Output An LI output in an agent a (of message c`!`v to agent b) is implemented simply by using a location-dependent (LD) output to send the message to channel `message` at the daemon, as an existential package with a triple $[b\ c\ v]$.

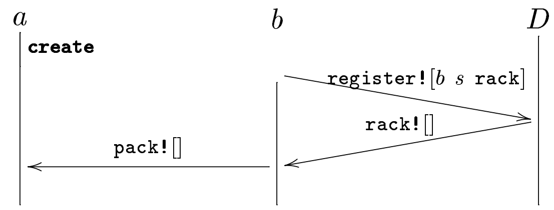
Reacting to the message on `message`, the daemon produces a message forwarding request $\mathsf{mesgReq}(\{T\}\,[b\ c\ v])$ (T being the type of v) which competes with other requests in acquiring the output on `lock`. Once successful, it looks up the target agent's site in the acquired site map, creates a fresh channel `dack` and forwards the message in LD mode (together with `dack`) to the `deliver` channel of b. In each agent, the `deliver` channel is handled by a `Deliverer` process, as in Figure 2.4. This reacts to `deliver` messages by emitting a local c`!`v message and acknowledging the daemon (again using LD communication) via `dack`. Meanwhile no agent may migrate before the `deliver` message arrives, since the daemon is single-threaded and waits for such an acknowledgement before releasing `lock`. Note that the **`notfound`** branch of the lookup will never be taken as the algorithm ensures that all agents register before messages can be sent to them. Following [SWP99], the inter-agent communications involved in the delivery of a single location-independent output are illustrated below.



Creation In order for the daemon's site map to be kept up-to-date, agents must register with the daemon, telling it their site. They must do this both upon creation and after migration. The encoding of **`create`**^{Z} $b = P$ **`in`** Q (in Figure 2.4) first acquires the local lock and current site s and then creates the new agent b, as well as fresh channels `pack` and `rack`. The body of b sends a `register` message to the daemon, supplying `rack`.

When *Daemon* at D receives a `register` message, it produces a registration request process $\mathsf{regReq}(b\ s\ \mathtt{rack})$ which competes with other requests in acquiring the output on `lock`. Upon success, the daemon updates the site map and then uses `rack` to acknowledge the registration of b. Meanwhile b may not begin executing before the acknowledgement arrives; this prevents b from sending a migration request to the daemon before it is registered.

After the acknowledgement is received from the daemon, b sends an acknowledgement to a using `pack`, initialises the local lock of b with s, installs a `Deliverer`, and allows the encoding of the body P of b to proceed. Meanwhile, the local lock and the encoding of the continuation process Q is kept until the acknowledgement via `pack` is received. This prevents the encoding of Q from using the daemon to forward LI messages to b before b is registered. The inter-agent communications involved in the creation of a single agent are illustrated below.

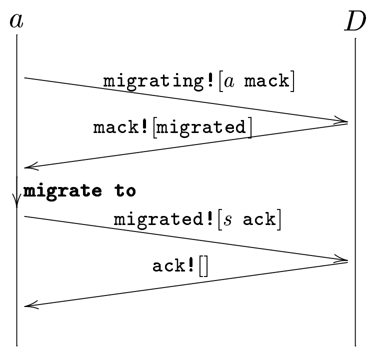


Migration The encoding of a migrating process in a (also in Figure 2.4) first acquires the output on `currentloc` at a (discarding the current site data). It then creates a fresh channel `mack` and sends a `migrating` message to the daemon with a tuple [a `mack`]. The encoding becomes migBlock(s `mack` P), waiting for the daemon to send a message to `mack` before proceeding with the migration. This ensures that if the daemon is sending a message on `deliver` channel to a (ie. forwarding an LI output), then such a message will arrive before a migrates away.

Reacting to the message on `migrating` message, *Daemon* produces a migrating request process migReq(a `mack`) which tries to acquire the lock and the current site map. Once it succeeds, it looks up the current site of a in the acquired map m, creating a fresh channel `migrated` and sending it (using an LD primitive) to a along channel `mack`. The request process at the daemon now becomes migProc(a m `migrated`), waiting for the requesting agent to complete its migration.

The blocked agent migBlock(s `mack` P), once it receives a message from `mack`, migrates agent a to the new site s, then creates a fresh channel `ack` and sends a tuple [s `ack`] to the daemon via channel `migrated` (using an LD primitive). Meanwhile, the local lock and the encoding of the continuation process P is kept until the acknowledgement via `ack` is received from the daemon.

Finally, the blocked daemon migProc(a m `migrated`) receives a message on `migrated`, updates the site map, then relinquishes the lock and then sends an acknowledgement to a at its new site. The inter-agent communications involved in the migration of a single agent are illustrated below.



Located Processes The definition of top-level encoding given in Equation (†) is restricted to certain forms of located processes. We may arrange an arbitrary located process into the desired form using the following translation.

$$
\begin{array}{rcl}
[\![@_a P]\!] & \stackrel{\text{def}}{=} & (\bullet; a \mapsto P) \\
[\![LP_1 \mid LP_2]\!] & \stackrel{\text{def}}{=} & (\Delta_1, \Delta_2; (\mathsf{A}_1 \boxplus \mathsf{A}_2)) \quad \text{where } [\![LP_i]\!] = (\Delta_i; \mathsf{A}_i) \\
[\![\mathbf{new}\ \Delta\ \mathbf{in}\ LP]\!] & \stackrel{\text{def}}{=} & (\Delta, \Theta; \mathsf{A}) \quad \text{where } [\![LP]\!] = (\Theta; \mathsf{A})
\end{array}
$$

$$
(\mathsf{A}_1 \boxplus \mathsf{A}_2)(a) \stackrel{\text{def}}{=} \begin{cases} \mathsf{A}_1(a) \mid \mathsf{A}_2(a) & a \in \mathsf{dom}(\mathsf{A}_1) \cap \mathsf{dom}(\mathsf{A}_2) \\ \mathsf{A}_1(a) & a \in \mathsf{dom}(\mathsf{A}_1) \wedge a \notin \mathsf{dom}(\mathsf{A}_2) \\ \mathsf{A}_2(a) & a \in \mathsf{dom}(\mathsf{A}_2) \wedge a \notin \mathsf{dom}(\mathsf{A}_1) \end{cases}
$$

It is not difficult to prove that indeed $LP \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ \prod_{a \in \mathsf{dom}(\mathsf{A})} \mathsf{A}(a)$ where $[\![LP]\!] = (\Delta; \mathsf{A})$.

Chapter 3

Type System

This chapter presents our type system for Nomadic π-calculi. We first give the typing judgements for different syntactic constructs in Section 3.1. Sections 3.3-3.6 give the axioms and inference rules for these typing judgements. We state some properties of the type system in Section 3.7. In Section 3.8, we define type-preserving substitutions, and prove that they preserve typability of processes. We also show that matching a value and a pattern of the same type yields a type-preserving substitution.

3.1 Forms of Typing Judgement

The richness of our type system (subtyping and polymorphism, in particular) leads us away from using the notion of sorting [Mil93b]. Taking into the consideration the rigorous programming language-like syntax of Nomadic π-calculi, we take as our starting point the style used in the definition of the programming language PICT [PT00]. We define *typing judgements* for different syntactic categories. The type system of Nomadic π-calculi consists of of axioms and inference rules defining sets of derivable typing judgements of the following forms:

$\vdash \Gamma$	unlocated type context Γ is well-formed.	(C-*)
$\vdash_{\mathsf{L}} \Gamma$	located type context Γ is well-formed.	(L-C-*)
$\Gamma \vdash S \leq T$	S is a subtype of T under assumptions Γ.	(SUBT-*)
$\Gamma \vdash T$	type T is well-formed under assumptions Γ.	(TYPE-*)
$\Gamma \vdash x@s$	the name x is located at s under assumptions Γ.	(VAR-LOC)
$\Gamma \vdash p \in T \triangleright \Delta$	pattern p requires type T and yields bindings Δ.	(PAT-*)

$\Gamma \vdash e \in T$	value or expression e has type T under assumptions Γ.	(Expr-*)
$\Gamma \vdash_a P$	basic process P as part of agent a is well-formed under assumptions Γ.	
$\Gamma \vdash LP$	located process LP is well-formed under assumptions Γ.	

In this thesis, type contexts are always located. Apart from the judgements of the form $\vdash_{\mathsf{L}} \Gamma$ and $\Gamma \vdash x@s$, however, we discard the site annotations in located type contexts, and work with unlocated type contexts when dealing with the above typing judgements. This avoids having to assign "dummy" annotations in typechecking pattern variables of type $\mathtt{Agent}^Z$ (see (Pat-Var)).

3.2 Subtyping Rules

The subtyping relation is built from the following order of capability tags:

$$\begin{tikzcd} \mathtt{r} \arrow[dr, dash] & & \mathtt{w} \arrow[dl, dash] & \mathtt{s} \arrow[d, dash] \\ & \mathtt{rw} & & \mathtt{m} \end{tikzcd}$$

The subtyping relation is defined by a rule for each type constructor or constant, as given in Figure 3.1.

(SubT-Base)
$$\frac{\Gamma \vdash B}{\Gamma \vdash B \leq B}$$

(SubT-Var)
$$\frac{\Gamma \vdash X}{\Gamma \vdash X \leq X}$$

(SubT-Site)
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{Site} \leq \mathtt{Site}}$$

(SubT-Agent)
$$\frac{\vdash \Gamma \quad Z \leq Z'}{\Gamma \vdash \mathtt{Agent}^Z \leq \mathtt{Agent}^{Z'}}$$

(SubT-Chan)
$$\frac{\begin{array}{ll} I = I' = \mathtt{rw} & \Rightarrow (\Gamma \vdash S \leq T \wedge \Gamma \vdash T \leq S) \\ I \leq I' = \mathtt{r} & \Rightarrow \Gamma \vdash S \leq T \\ I \leq I' = \mathtt{w} & \Rightarrow \Gamma \vdash T \leq S \end{array}}{\Gamma \vdash \hat{}^I S \leq \hat{}^{I'} T}$$

(SubT-Exist)
$$\frac{\Gamma, X \vdash S \leq T}{\Gamma \vdash \{X\}\, S \leq \{X\}\, T}$$

(SubT-Tuple)
$$\frac{\Gamma \vdash S_1 \leq T_1 \quad \ldots \quad \Gamma \vdash S_n \leq T_n}{\Gamma \vdash [S_1 \ldots S_n] \leq [T_1 \ldots T_n]}$$

Figure 3.1: Rules for subtyping

Base types, site type and type variables are simply subtypes of themselves (SubT-Base, SubT-Var and SubT-Site). Agent subtyping is simple as `Agent` is simply a type not a constructor. Since mobile agents are more capable than static agents, we take $\mathtt{Agent}^{\mathtt{m}} \leq \mathtt{Agent}^{\mathtt{s}}$ (SUBT-AGENT).

The rule (SUBT-CHAN) shows the channel constructor $\mathtt{\hat{}}^{\mathtt{r}}$ covariant in its argument, whereas $\mathtt{\hat{}}^{\mathtt{w}}$ contravariant, as in [PS96]. Operationally, this captures the observation that: if a given channel c is used only to read elements of type T then it is safe to replace c by another channel c' carrying elements of type S, as long as any element that is read from c' may safely be regarded as an element of T — that is, as long as S is a subtype of T. The constructor $\mathtt{\hat{}}^{\mathtt{rw}}$ is invariant in the subtype relation (i.e. $\mathtt{\hat{}}^{\mathtt{rw}}S$ is a subtype of $\mathtt{\hat{}}^{\mathtt{rw}}T$ only when S and T are equivalent — ie. identical in our calculi). The type $\mathtt{\hat{}}^{\mathtt{rw}}T$ is a subtype of both $\mathtt{\hat{}}^{\mathtt{r}}T$ and $\mathtt{\hat{}}^{\mathtt{w}}T$. That is, we are allowed to forget either the capability to write or the capability to read on a channel: a channel that can be used for both input and output may be used in a context where just one capability is needed.

Other type constructors are covariant. A tuple S is a subtype of another tuple T whenever S and T contain exactly the same number of elements and the types of the corresponding elements are also subtypes (SUBT-TUPLE). An existential type $\{X\}\, S$ is a subtype of of $\{X\}\, T$ if S is a subtype of T, with X being a part of the assumption (SUBT-EXIST).

3.3 Type and Type Context Formation

The rules for unlocated type context formation are given in Figure 3.2. They ensure that none of the declared type variables and names are duplicated, and that each type in the binding is well-formed. To typecheck located type contexts, we also need to ensure the validity of site annotations. The rules for located type context formation are similar to those for unlocated type contexts, replacing the rule (C-VAR) by (L-C-VAR) and (L-C-AGENT). Such rules are given in Figure 3.3.

$$\text{(C-EMPTY)}\quad \vdash \bullet \qquad\qquad \text{(C-TVAR)}\quad \frac{\vdash \Gamma \quad X \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma,\ X} \qquad\qquad \text{(C-VAR)}\quad \frac{\vdash \Gamma \quad \Gamma \vdash T \quad x \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma,\ x : T}$$

Figure 3.2: Rules for unlocated type context formation

The rules for type formation are given in Figure 3.4. They are standard; we omit their explanation.

(L-C-Empty)
$$\vdash_{\mathsf{L}} \bullet$$

(L-C-TVar)
$$\frac{\vdash_{\mathsf{L}} \Gamma \quad X \notin \mathsf{dom}(\Gamma)}{\vdash_{\mathsf{L}} \Gamma,\ X}$$

(L-C-Var)
$$\frac{\vdash_{\mathsf{L}} \Gamma \quad T \neq \mathtt{Agent}^Z \quad \Gamma \vdash T \quad x \notin \mathsf{dom}(\Gamma)}{\vdash_{\mathsf{L}} \Gamma,\ x : T}$$

(L-C-Agent)
$$\frac{\vdash_{\mathsf{L}} \Gamma \quad \Gamma \vdash s \in \mathtt{Site} \quad \Gamma \vdash \mathtt{Agent}^Z \quad x \notin \mathsf{dom}(\Gamma)}{\vdash_{\mathsf{L}} \Gamma,\ x : \mathtt{Agent}^Z@s}$$

Figure 3.3: Rules for located type context formation

(Type-Base)
$$\frac{\vdash \Gamma \quad B \in \mathcal{T}}{\Gamma \vdash B}$$

(Type-Site)
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{Site}}$$

(Type-Agent)
$$\frac{\vdash \Gamma \quad Z \in \{\mathtt{m}, \mathtt{s}\}}{\Gamma \vdash \mathtt{Agent}^Z}$$

(Type-Exist)
$$\frac{\Gamma, X \vdash T \quad X \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \{X\}\, T}$$

(Type-Var)
$$\frac{\vdash \Gamma \quad X \in \mathsf{dom}(\Gamma)}{\Gamma \vdash X}$$

(Type-Chan)
$$\frac{\Gamma \vdash T \quad I \in \{\mathtt{r}, \mathtt{w}, \mathtt{rw}\}}{\Gamma \vdash \hat{}^{I} T}$$

(Type-Tuple)
$$\frac{\Gamma \vdash T_1 \quad \dots \quad \Gamma \vdash T_n}{\Gamma \vdash [T_1 \dots T_n]}$$

Figure 3.4: Rules for type formation

3.4 Values and Expressions

Due to their shared structure, the same set of judgement rules are used for both values and expressions. We let e range over values *and* expressions. The rules are given in Figure 3.5.

$$\text{(VAR-ID)}\quad \frac{\vdash \Gamma,\ x : T,\ \Gamma'}{\Gamma,\ x : T,\ \Gamma' \vdash x \in T} \qquad \text{(EXPR-BVAL)}\quad \frac{\vdash \Gamma \quad t \in B}{\Gamma \vdash t \in B} \qquad \text{(EXPR-SUB)}\quad \frac{\Gamma \vdash e \in S \quad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T}$$

$$\text{(EXPR-EXIST)}\quad \frac{\Gamma \vdash e \in \{S/X\}T \quad \Gamma \vdash S \quad \Gamma, X \vdash T}{\Gamma \vdash \{S\}\, e \in \{X\}\, T} \qquad \text{(EXPR-TUPLE)}\quad \frac{\Gamma \vdash e_1 \in T_1 \quad \dots \quad \Gamma \vdash e_n \in T_n}{\Gamma \vdash [e_1 \dots e_n] \in [T_1 \dots T_n]}$$

$$\text{(EQUALITY)}\quad \frac{\Gamma \vdash e \in T \quad \Gamma \vdash e' \in T}{\Gamma \vdash e = e' \in \texttt{Bool}}$$

$$\text{(EVAL)}\quad \frac{f \in \mathcal{F} \quad f \texttt{ is not } = \quad f :\ B_1 \times \ldots \times B_n \to B \quad \Gamma \vdash e_1 \in B_1 \ \ldots\ \Gamma \vdash e_n \in B_n}{\Gamma \vdash f(e_1, \ldots, e_n) \in B}$$

Figure 3.5: Rules for value and expression formation

If the type context contains the binding $x : T$ then the type of x is T in this context (VAR-ID) (all binding names are unique — ensured by the precondition, so there is no ambiguity in this rule). Constants of base types simply possess such types (EXPR-BVAL).

The subsumption rule (EXPR-SUB) allows types of values to be *promoted* in the subtype relation: if e is a value of type S and S is a subtype of T then e is also of type T.

A expression $\{S\}\, e$ is an existential package of type $\{X\}\, T$ if the "witness type" S is well-formed, and the actual type of the expression e must match the type T after the substitution of S for X. Readers familiar with typed λ-calculi will recognise the similarity of this rule to the standard introduction rule for existential types (cf. [MP88]). We prevent the escape of the hidden type variable rule by the second and third assumptions of (EXPR-EXIST), ensuring that X is not in the domain of Γ and hence not free in S.

If the expressions e_1 through e_n have the type T_1 through T_n then the tuple expression $[e_1 \dots e_n]$ has the tuple type $[T_1 \dots T_n]$ (EXPR-TUPLE). To typecheck a function application expression $f(e_1, \ldots, e_n)$, we look up the type of the function f, typecheck each argument expression according to the argument type obtained, and the type of the whole expression is the type of the range of f (EVAL, EQUALITY). As discussed in Section 2.2.3 on page 21, with an exception of equality test, where the arguments can be of any type, the type of each argument as well as the result must be a base type.

(VAR-LOC)
$$\frac{\vdash_{\mathsf{L}} \Gamma,\ x : \mathtt{Agent}^{Z}@s,\ \Gamma'}{\Gamma,\ x : \mathtt{Agent}^{Z}@s,\ \Gamma' \vdash x@s}$$

Figure 3.6: Rule for locating names

The location of an agent can be looked up using (VAR-LOC) in Figure 3.6. This judgement form $\Gamma \vdash a@s$ is not employed in typechecking, but is essential in the operational semantics.

3.5 Patterns

Pattern typing statements have the form $\Gamma \vdash p \in T \triangleright \Delta$, ie. each pattern has a type, describing the values it can match, and moreover gives rise to a set of extra bindings Δ. Their rules are given in Figure 3.7.

(PAT-VAR)
$$\frac{\vdash \Gamma \quad \Gamma \vdash T}{\Gamma \vdash x \in T \triangleright x : T}$$

(PAT-TUPLE)
$$\frac{\Gamma \vdash p_1 \in T_1 \triangleright \Delta_1 \quad \ldots \quad \Gamma \vdash p_n \in T_n \triangleright \Delta_n}{\Gamma \vdash [p_1 \ldots p_n] \in [T_1 \ldots T_n] \triangleright \Delta_1, \ldots, \Delta_n}$$

(PAT-WILD)
$$\frac{\vdash \Gamma \quad \Gamma \vdash T}{\Gamma \vdash \_ \in T \triangleright \bullet}$$

(PAT-EXIST)
$$\frac{\Gamma, X \vdash p \in T \triangleright \Delta \quad X \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \{X\}\, p \in \{X\}\, T \triangleright X, \Delta}$$

Figure 3.7: Rules for pattern formation

A variable pattern x and a wildcard pattern _ requiring type T match any value of type T (provided that T is a well-formed type) and gives rise to, in the case of variable pattern, a binding for the variable x (PAT-VAR), and, in the case of wildcard pattern, no binding (PAT-WILD).

A tuple pattern $[p_1 \ldots p_n]$ requiring type $[T_1 \ldots T_n]$ is typable if each subpattern p_i requiring type T_i is typable; moreover, it gives rise to a set of bindings comprising all the bindings from its subpatterns (PAT-TUPLE).

An existential pattern $\{X\}\, p$ matches any value of type $\{X\}\, T$, where T is the type required by the pattern p (under the assumed existence of X). Such a pattern yields not only the bindings produced by p, but also the "existence of X" binding (PAT-EXIST).

3.6 Basic and Located Process

The judgement rules for basic and located processes are relatively simple. Valid formation of a basic process, in addition to ensuring correct deployment of names, must also consider its ability to be located at agents — in particular, if such a process contains any possibility of migration, it cannot be located in a static agent. We give the judgement rules for basic process in Figures 3.8-3.11.

(REP-IN)
$$\frac{\Gamma \vdash x \in \texttt{\^{}r}T \quad \Gamma \vdash p \in T \rhd \Delta \quad \Gamma, \Delta \vdash_a P}{\Gamma \vdash_a x\texttt{?}p \rightarrow P \quad \text{and} \quad \Gamma \vdash_a \texttt{*}x\texttt{?}p \rightarrow P}$$

(OUT)
$$\frac{\Gamma \vdash a \in \texttt{Agent}^{\texttt{s}} \quad \Gamma \vdash c \in \texttt{\^{}w}T \quad \Gamma \vdash v \in T}{\Gamma \vdash_a c\texttt{!}v}$$

(SENDLI)
$$\frac{\Gamma \vdash a, b \in \texttt{Agent}^{\texttt{s}} \quad \Gamma \vdash c \in \texttt{\^{}w}T \quad \Gamma \vdash v \in T}{\Gamma \vdash_a \langle b@?\rangle c\texttt{!}v}$$

Figure 3.8: Rules for basic processes: input and output

A (replicated) input process $c\texttt{?}p \rightarrow P$ (respectively $\texttt{*}c\texttt{?}p \rightarrow P$) is well-formed if c is a channel which can be used for input, ie. its type can be promoted to $\texttt{\^{}r}T$ for some T. The guarded process P must be well-typed with respect to the context extended with the bindings Δ, introduced by typing pattern p requiring type T (REP-IN). The typechecking rules for output processes, including the sugared LD output and LI output, are similar to each other. The message $c\texttt{!}v$ is well-typed if c can be used for output, ie. its type can be promoted to $\texttt{\^{}w}T$ for some T matching the type of v (OUT). In case of LI outputs, we must also ensure that the agent target of the message is really an agent (SENDLI). Note that, by using the subsumption rule (EXPR-SUB), a mobile agent may have type $\texttt{Agent}^{\texttt{s}}$. The assumption $\Gamma \vdash a \in \texttt{Agent}^{\texttt{s}}$ is used whenever we need to ensure that a is an agent.

A conditional expression is well-formed if the guard expression has boolean type and both branches of the conditional are well-formed (COND). A let declaration **let** $p = ev$ **in** P is well-formed if the expression is of a type T and the guarded process P is well-typed with respect to the context extended with the bindings Δ, introduced by typing pattern p requiring type T (LET).

(COND)
$$\frac{\Gamma \vdash v \in \texttt{Bool} \quad \Gamma \vdash_a P \quad \Gamma \vdash_a Q}{\Gamma \vdash_a \textbf{if } v \textbf{ then } P \textbf{ else } Q}$$

(LET)
$$\frac{\Gamma \vdash ev \in T \quad \Gamma \vdash p \in T \triangleright \Delta \quad \Gamma, \Delta \vdash_a P}{\Gamma \vdash_a \textbf{let } p = ev \textbf{ in } P}$$

Figure 3.9: Rules for basic processes: internal computation

A **create**^{Z} $b = P$ **in** Q expression is well-formed as part of an agent a if b is fresh and, assuming b of type $\texttt{Agent}^Z$, P is well-typed as part of b and Q is well-typed as part of a (CREATE). Typechecking **migrate to** $s \to P$ as part of agent a involves checking that agent a is capable of migrating (ie. it has type $\texttt{Agent}^{\texttt{m}}$), that s is a site and that P is well-formed (MIGRATE). Similarly to the conditional, an **iflocal** expression is well-formed if the guard expression (which can be regarded as an LD output) and both branches are well-formed (IFLOCAL).

(CREATE)
$$\frac{a \neq b \quad \Gamma,\ b : \texttt{Agent}^Z \vdash_b P \quad \Gamma,\ b : \texttt{Agent}^Z \vdash_a Q}{\Gamma \vdash_a \textbf{create}^Z\ b = P \textbf{ in } Q}$$

(MIGRATE)
$$\frac{\Gamma \vdash a \in \texttt{Agent}^{\texttt{m}} \quad \Gamma \vdash s \in \texttt{Site} \quad \Gamma \vdash_a P}{\Gamma \vdash_a \textbf{migrate to } s \to P}$$

(IFLOCAL)
$$\frac{\Gamma \vdash b \in \texttt{Agent}^{\texttt{s}} \quad \Gamma \vdash c \in \texttt{\^{}w}T \quad \Gamma \vdash v \in T \quad \Gamma \vdash_a P \quad \Gamma \vdash_a Q}{\Gamma \vdash_a \textbf{iflocal } \langle b \rangle c\texttt{!}v \textbf{ then } P \textbf{ else } Q}$$

Figure 3.10: Rules for basic processes: agent constructs

A **0** process locating at a is always well-typed (NIL), provided that the type context is well-formed and that a is indeed an agent. The parallel composition of two basic processes is well-formed with respect to a given context if each process is (PAR). A **new** declaration is well-typed if the process under such declaration is well-typed with respect to the assumption extended with the new binding (LOCALNEW).

(NIL)
$$\frac{\vdash \Gamma \quad \Gamma \vdash a \in \texttt{Agent}^{\texttt{s}}}{\Gamma \vdash_a \mathbf{0}}$$

(PAR)
$$\frac{\Gamma \vdash_a P \quad \Gamma \vdash_a Q}{\Gamma \vdash_a P \mid Q}$$

(LOCALNEW)
$$\frac{\Gamma,\ x : \texttt{\^{}}^I T \vdash_a P}{\Gamma \vdash_a \textbf{new}\ x : \texttt{\^{}}^I T\ \textbf{in}\ P}$$

Figure 3.11: Rules for basic processes: composition

Located Process Due to the simplicity of the located process syntax, the typechecking rules for located processes (given in Figure 3.12) are straightforward. A located basic process $@_a P$ is well-formed if a is an agent and P is well-formed as part of a (AT). The rules for parallel composition and channel declaration are similar to those defined for their basic process counterparts (LPAR) and (NEWCHANNEL). Agent declaration is similar to channel declaration, although we also need to make sure that the site annotation attached is indeed a site (NEWAGENT).

(AT)
$$\frac{\Gamma \vdash_a P \quad \Gamma \vdash a \in \texttt{Agent}^{\texttt{s}}}{\Gamma \vdash @_a P}$$

(LPAR)
$$\frac{\Gamma \vdash LP \quad \Gamma \vdash LQ}{\Gamma \vdash LP \mid LQ}$$

(NEWCHANNEL)
$$\frac{\Gamma,\ c : \texttt{\^{}}^I T \vdash LP}{\Gamma \vdash \textbf{new}\ c : \texttt{\^{}}^I T\ \textbf{in}\ LP}$$

(NEWAGENT)
$$\frac{\Gamma,\ a : \texttt{Agent}^Z \vdash LP \quad \Gamma \vdash s \in \texttt{Site}}{\Gamma \vdash \textbf{new}\ a : \texttt{Agent}^Z @ s\ \textbf{in}\ LP}$$

Figure 3.12: Rules for located process formation

3.7 Basic Properties

We have proved several properties of this type system, including the property that it is preserved by valid type context permutation (Lemma 3.7.1), and strengthening and weakening of type contexts (Lemma 3.7.2). We also prove that, whenever evaluation function $\texttt{eval}(\cdot)$ is defined, it preserves types. These properties are crucial for the subsequent lemmas, such as the subject reduction (Theorem 4.5.1), as well as those concerning operational relations. The proofs of these lemmas are routine inductions on type derivation; we omit their details.

Lemma 3.7.1 (Context permutation preserves typing)

1. If $\vdash_{\mathsf{L}} \Gamma$ and $\Delta \equiv \Gamma$ then $\vdash_{\mathsf{L}} \Delta$.

2. If $\Gamma \vdash_a P$ and $\Delta \equiv \Gamma$ then $\Delta \vdash_a P$.

3. If $\Gamma \vdash LP$ and $\Delta \equiv \Gamma$ then $\Delta \vdash LP$.

Lemma 3.7.2 (Typing: weakening and strengthening)

1. $\Gamma \vdash e \in T$ and $\vdash \Gamma, \Delta$ IFF $\Gamma, \Delta \vdash e \in T$ and $\mathsf{dom}(\Delta) \cap \mathsf{fv}(e) = \emptyset$.

2. $\Gamma \vdash p \in T \triangleright \Theta$, $\vdash \Gamma, \Delta$ and $\mathsf{dom}(\Delta) \cap \mathsf{fv}(p) = \emptyset$ IFF $\Gamma, \Delta \vdash p \in T \triangleright \Theta$ and $\mathsf{dom}(\Delta) \cap (\mathsf{fv}(p) \cap \mathsf{fv}(T)) = \emptyset$.

3. $\Gamma \vdash_a P$ and $\vdash \Gamma, \Delta$ IFF $\Gamma, \Delta \vdash_a P$ and $\mathsf{dom}(\Delta) \cap (\mathsf{fv}(P) \cup \{a\}) = \emptyset$.

4. $\Gamma \vdash LP$ and $\vdash \Gamma, \Delta$ IFF $\Gamma, \Delta \vdash LP$ and $\mathsf{dom}(\Delta) \cap \mathsf{fv}(LP) = \emptyset$.

Lemma 3.7.3 (Evaluation preserves types)
If $\Gamma \vdash e \in T$ and $\mathtt{eval}(e)$ is defined then $\Gamma \vdash \mathtt{eval}(e) \in T$.

3.8 Type-Preserving Substitution and Matching

Substitutions, ranged over by σ, ρ, are finite maps associating names with values, as well as type variables with types. Substitution is extended in the natural way to an operation on processes (renaming bound names as necessary to avoid capture). We write

$$\{T_1/X_1, \ldots, T_n/X_n, v_1/x_1, \ldots, v_k/x_k\}$$

for the substitution that simultaneously replaces every occurrences of a type variable X_i by the type T_i, and a name x_j by the value v_j. The result of applying σ to a process P is written as σP, and similarly for other constructs such as type contexts, patterns and values. We assign substitutions the highest precedence over the operators of the calculus. If a name substitution is injective, it can be used for renaming free names in a process, often to avoid name clashing. We prove that typing is preserved by renaming.

Lemma 3.8.1 (Injective substitution preserves typing)
Given that σ is an injective name substitution,

1. if $\Gamma \vdash_a P$ then $\sigma\Gamma \vdash_{\sigma a} \sigma P$; and

2. if $\Gamma \vdash LP$ then $\sigma\Gamma \vdash \sigma LP$.

In a strongly typed setting, it is important that name substitutions are type-preserving, formally defined as follows:

Definition 3.8.1 (Type-Preserving Substitution)
A substitution σ is said to be *type-preserving* w.r.t. Γ if,

- for all $x \in \mathsf{dom}(\sigma)$, $\Gamma \vdash x \in T$ implies $\Gamma \vdash \sigma x \in \sigma T$; and
- for all $X \in \mathsf{dom}(\sigma)$, $\Gamma \vdash X$ implies $\Gamma \vdash \sigma X$.

Since our type system is polymorphic, a type-preserving substitution ensures that when substituting a name x of type T, the value substituted to x has the type T after the occurrences of type variables have been resolved. For example, if an input process $c\mathbf{?}\,\{X\}\,[x] \rightarrow P$, well-typed w.r.t. Γ, receives a value $\{\mathtt{Int}\}\,[5]$ along c, then the substitution $\sigma = \{\mathtt{Int}/X,\ 5/x\}$ will be applied to P. In this case, σ is type-preserving w.r.t. $\Gamma, X, x : X$.

Type-preserving substitutions indeed preserve process typing. The formal statement of this is given below.

Lemma 3.8.2 (Type-preserving substitution preserves typing)
If a substitution σ is type-preserving w.r.t. Γ then

1. $\Gamma \vdash_a P$ implies $\Gamma \vdash_{\sigma a} \sigma P$; and
2. $\Gamma \vdash LP$ implies $\Gamma \vdash \sigma LP$.

Proof: Routine induction on the typing derivation of $\Gamma \vdash_a P$ and of $\Gamma \vdash LP$. ■

When an input process receives a value v along a channel, it needs to deconstruct v, producing a substitution to be applied to its continuation process. This can be done using *matching*: a partial function mapping pairs of patterns and values to name substitutions, whenever they are of the same shape. Its formal definition is given in Figure 3.13.

$$
\begin{array}{lcl}
\mathsf{match}(\_, v) & \stackrel{\text{def}}{=} & \{\} \\
\mathsf{match}(x, v) & \stackrel{\text{def}}{=} & \{v/x\} \\
\mathsf{match}([p_1 \ldots p_n], [v_1 \ldots v_n]) & \stackrel{\text{def}}{=} & \mathsf{match}(p_1, v_1) \cup \ldots \cup \mathsf{match}(p_n, v_n) \\
\mathsf{match}(\{X\}\, p, \{T\}\, v) & \stackrel{\text{def}}{=} & \{T/X\} \ \cup\ \mathsf{match}(p, v) \\
\mathsf{match}(p, v) & \stackrel{\text{def}}{=} & \perp \ \text{(undefined)} \quad \text{otherwise}
\end{array}
$$

Figure 3.13: Matching

The matching function simultaneously traverses the structure of the pattern and the value, yielding bindings when variables are encountered in the pattern. Note that the variables bound in a pattern are always distinct, so the $\cup$ operation, in effect combining the substitutions with *disjoint* domains, is always well-defined.

In a typed semantics, the type system can prevent a mismatch between the value received and the shape expected in communication. However, matching a value and a pattern of the same type does not always yield a substitution: for example let Γ be $x : [[]\ []]$, a pattern $[y\ z]$ may have type $[[]\ []]$ w.r.t. Γ, but $\mathsf{match}([y\ z], x)$ is undefined. Similar situation occurs when matching a name x of an existential type to an existential pattern $\{X\}\, p$. To prevent this, we define *ground* type contexts as follows.

Definition 3.8.2 (Ground Type Context)
A type context Γ is ground if, for all $x \in \mathsf{dom}(\Gamma)$, $\Gamma \vdash x \in T$ implies $T \neq [T_1 \ldots T_n]$ and $T \neq \{X\}\, S$, for any $T_1, \ldots, T_n, X, S$.

Ground type contexts ensure that a pattern and a value of the same type can be matched, producing a type-respecting substitution. This result is crucial to the proof of subject reduction (Theorem 4.5.1), its formal statement is given below.

Lemma 3.8.3 (Ground type context ensures matching is type-preserving)
Given that Γ is a ground located type context, if $\Gamma \vdash v \in S$ and $\Gamma \vdash p \in S \triangleright \Xi$ then $\mathsf{match}(p, v)$ is defined, and is a type-preserving substitution w.r.t. Γ, Ξ.

Proof: The proof uses an induction on typing derivations of $\Gamma \vdash v \in S$. By insisting that Γ is ground, the pattern p may have a tuple type if and only if the value v is a tuple; and similarly for p being of existential type.

The details may be found in Appendix B.1 on page 190. ■

Chapter 4

Operational Semantics

This chapter makes precise the informal descriptions of Nomadic π primitives given in Section 2.2.4 by giving two operational semantics: a reduction semantics and a labelled transition semantics. The former formally describes the evolution of processes via internal computation. This builds on the reduction semantics of Nomadic π-calculi given in [SWP99], adding a type system. In order to do compositional reasoning, however, the reduction semantics is insufficient. For that we define a labelled transition semantics (LTS), expressing how processes interact with their environment.

The basic semantic theory for a π-calculus has long been associated with the aforementioned styles of operational semantics. The reduction semantics, building on the Chemical Abstract Machine ideas of Berry and Boudol [BB92] and the π semantics of Milner [Mil93b], are relatively easy to define, and are widely employed for capturing informal intuition in novel calculi. This style of semantics involves defining two relations on processes: a reduction relation, which formalises how processes perform internal computations, and a structural congruence relation. The structural congruence relation allows us to rewrite process terms so that eg. any two co-located active input or output prefixes can be syntactically juxtaposed. The labelled transition semantics, on the other hand, is more suited to compositional reasoning, since it clarifies how processes interact (typically by inputs and outputs) with the environment. Moreover, the labelled transitions of process terms are defined inductively on its syntactic structure and hence do not rely on structural congruence. In general, the existence of some particular reduction is easier to show using the reduction semantics, whereas all possible reductions are easier to enumerate by means of the labelled transition semantics.

Adapting these semantics for Nomadic π-calculi involves extending the standard π-calculus reduction and labelled transition semantics to deal with agent mobility, inter-agent communication, and a rich type system. We argue in Section 4.3 that migration of a free agent must be considered an observable action in the labelled transition semantics, since migration of an agent causes the location map to be updated — an action which has a direct effect on some location-dependent actions performed by agents in the environment.

In this thesis, we are working with *typed* semantics, where the reduction/transition relations are defined over well-typed process terms and, for the labelled semantics, allowing only well-typed inputs and outputs with the environment. We adopt this for the following reasons.

- Typed semantics give a tighter notion of behaviour, definitely excluding pathological terms whose behaviour is undesirable or even impossible to implement, eg. $@_a a\,\mathtt{!}\,[]$.
- Typed semantics simplifies the statement of subject reduction (Theorem 4.5.1) and, in the labelled transition semantics, it allows the types of names (and the current sites of agents) extruded by input or output actions to be made precise. The alternative, of untyped semantics, has simpler definition of the rules, but more complex subject reduction. It would be required where one wishes to consider interaction with badly-typed processes, eg. a malicious attacker.

Before giving the formal definition of the semantics, we define structural congruence in Section 4.1. The reduction semantics and the labelled transition semantics are then given in Section 4.2 and Section 4.3. The latter section also discusses some alternative styles of labelled transition semantics, and why we discard them. Section 4.4 discusses runtime errors, and how the type system guarantees the absence of such errors. We conclude this chapter by stating properties of our semantics, including the subject reduction and reduction/LTS correspondence.

4.1 Structural Congruence

We define a *structural congruence* as the smallest congruence relation closed under alpha-conversion and the rules in Figures 4.1-4.2. The relation is defined for both basic and located processes, denoted $P \equiv Q$ and $LP \equiv LQ$.

The structural congruence rules are similar to a standard structural congruence for an asynchronous π-calculus, with scope extrusion both for the new channel binder $\mathbf{new}\ c : \hat{\ }^I T\ \mathbf{in}\ P$ (Str-Extrude) and the new agent binder $\mathbf{new}\ a : \mathtt{Agent}^Z@s\ \mathbf{in}\ LP$ (Str-L-Extrude). Two rules given in Figure 4.2, however, are specific for this setting. The rules (Str-Distr)

(STR-NULL) $P \equiv P \mid \mathbf{0}$

(STR-PRL) $P \mid Q \equiv Q \mid P$

(STR-ASSOC) $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

(STR-EXTRUDE)
$$\frac{x \notin \mathsf{fv}(P)}{P \mid \mathtt{new}\ x : T\ \mathtt{in}\ Q \equiv \mathtt{new}\ x : T\ \mathtt{in}\ (P \mid Q)}$$

(STR-SWAPNEW)
$$\frac{x, x' \text{ are distinct}}{\mathtt{new}\ x : T,\ x' : T'\ \mathtt{in}\ P \equiv \mathtt{new}\ x' : T',\ x : T\ \mathtt{in}\ P}$$

Figure 4.1: Structural congruence: Basic processes

and (STR-N-EXTRUDE) distribute location annotations down to basic process terms. More precisely, (STR-DISTR) enables parts of an agent to be syntactically separated or brought together, and (STR-N-EXTRUDE) allows channel binders to be extruded past locators.

Note that we choose not to include the null process in the syntax of located process; an inactive located process is written $@_a\mathbf{0}$ rather than simply $\mathbf{0}$. A reason for this is to avoid a structural congruence rule $@_a\mathbf{0} \equiv \mathbf{0}$, which does not preserve free variables. A null process located at a can be garbage-collected if it is placed in parallel with a located process which contains fragments of agents a. For example, it is possible to derive $LP \equiv LP|@_a\mathbf{0}$ if $LP = @_aP|LQ$, but not possible if $a \notin \mathsf{fv}(LP)$.

We prove the following properties for the structural congruence: that it preserves free variables as well as typing, and that it is preserved by arbitrary name substitution. Note in Lemma 4.1.3 that the size of the derivation of a structural congruence relation is preserved by name substitution; this is essential for proofs involving an induction on the size of the derivation of $LP \equiv LQ$, eg. that of Theorem 4.5.2. The proofs of these lemmas are routine inductions on the derivation of $P \equiv Q$ and $LP \equiv LQ$; we omit the details.

Lemma 4.1.1 (Preservation of free names under str. cong.)

1. If $P \equiv Q$ then $\mathsf{fv}(P) = \mathsf{fv}(Q)$.

2. If $LP \equiv LQ$ then $\mathsf{fv}(LP) = \mathsf{fv}(LQ)$.

Lemma 4.1.2 (Type soundness of str. cong.)

1. If $P \equiv Q$ and $\Gamma \vdash_a P$ then $\Gamma \vdash_a Q$.

2. If $LP \equiv LQ$ and $\Gamma \vdash LP$ then $\Gamma \vdash LQ$.

(Str-Distr)
$$@_a (P \mid Q) \equiv @_a P \mid @_a Q$$

(Str-L-Assoc)
$$LP \mid (LQ \mid LR) \equiv (LP \mid LQ) \mid LR$$

(Str-L-Prl)
$$LP \mid LQ \equiv LQ \mid LP$$

(Str-N-Extrude)
$$\frac{x \neq a}{@_a(\mathbf{new}\ x : \hat{}^I T\ \mathbf{in}\ P) \equiv \mathbf{new}\ x : \hat{}^I T\ \mathbf{in}\ @_a P}$$

(Str-Locate)
$$\frac{P \equiv Q}{@_a P \equiv @_a Q}$$

(Str-L-SwapNew)
$$\frac{\Delta \equiv \Xi}{\mathbf{new}\ \Delta\ \mathbf{in}\ LP \equiv \mathbf{new}\ \Xi\ \mathbf{in}\ LP}$$

(Str-L-Extrude)
$$\frac{x \notin \mathsf{fv}(LP)}{\begin{array}{l} LP \mid \mathbf{new}\ x : T\ \mathbf{in}\ LQ \equiv \mathbf{new}\ x : T\ \mathbf{in}\ (LP \mid LQ) \\ LP \mid \mathbf{new}\ x : \mathtt{Agent}^Z @s\ \mathbf{in}\ LQ \equiv \mathbf{new}\ x : \mathtt{Agent}^Z @s\ \mathbf{in}\ (LP \mid LQ) \end{array}}$$

Figure 4.2: Structural congruence: Located processes

Lemma 4.1.3 (Structural congruence is preserved under name substitution)

1. For any derivation of $P \equiv Q$, there is a derivation of the same size of $\rho P \equiv \rho Q$, for any substitution ρ.

2. For any derivation of $LP \equiv LQ$, there is a derivation of the same size of $\rho LP \equiv \rho LQ$, for any substitution ρ.

4.2 Reduction Semantics

In this section, we define the reduction semantics of Nomadic π-calculi as a reduction relation between *configurations*, which are pairs $\Gamma \Vdash LP$ of a located type context Γ and a located process LP. We are concerned only with configurations for well-typed programs, and define a reduction relation $\rightarrow$ as the smallest relation from $\{\Gamma \Vdash LP \mid \ \vdash_{\mathsf{L}} \Gamma \text{ and } \Gamma \vdash LP\}$ to $\{\Gamma \Vdash LP \mid LP \in \mathrm{n}\pi_{\mathsf{LD,LI}}\}$, closed under the rules given in Figures 4.3-4.4.

The informal description of the primitives given in Section 2.2.4 should explain these rules, so we omit detailed description here. The semantics of the high-level calculus is obtained by adding (Red-LI-Send), for delivering LI messages to their destination agent.

(RED-CREATE)

$$\frac{\Gamma \vdash a@s}{\Gamma \Vdash @_a\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q \ \rightarrow\ \Gamma \Vdash \mathbf{new}\ b : \mathtt{Agent}^Z@s\ \mathbf{in}\ (@_b P \mid @_a Q)}$$

(RED-MIGRATE)

$$\Gamma \Vdash @_a\mathbf{migrate\ to}\ s \rightarrow P \ \rightarrow\ (\Gamma \oplus a \mapsto s) \Vdash @_a P$$

(RED-IFLOCAL-TRUE)

$$\frac{\Gamma \vdash a@s \quad \Gamma \vdash b@s}{\Gamma \Vdash @_a\mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ P\ \mathbf{else}\ Q \ \rightarrow\ \Gamma \Vdash @_a P \mid @_b c!v}$$

(RED-IFLOCAL-FALSE)

$$\frac{\Gamma \vdash a@s \quad \Gamma \vdash b@s' \quad s \neq s'}{\Gamma \Vdash @_a\mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ P\ \mathbf{else}\ Q \ \rightarrow\ \Gamma \Vdash @_a Q}$$

(RED-LET)

$$\frac{\mathtt{eval}(ev)\ \text{defined}}{\Gamma \Vdash @_a\mathbf{let}\ p = ev\ \mathbf{in}\ P \ \rightarrow\ \Gamma \Vdash @_a\mathsf{match}(p, \mathtt{eval}(ev))\ P}$$

(RED-COND-TRUE)

$$\Gamma \Vdash @_a\mathbf{if\ true\ then}\ P\ \mathbf{else}\ Q \ \rightarrow\ \Gamma \Vdash @_a P$$

(RED-COND-FALSE)

$$\Gamma \Vdash @_a\mathbf{if\ false\ then}\ P\ \mathbf{else}\ Q \ \rightarrow\ \Gamma \Vdash @_a Q$$

(RED-COMM)

$$\Gamma \Vdash @_a\,(c!v|c?p \rightarrow P) \ \rightarrow\ \Gamma \Vdash @_a\mathsf{match}(p, v)P$$

(RED-REPLIC)

$$\Gamma \Vdash @_a\,(c!v|{*}c?p \rightarrow P) \ \rightarrow\ \Gamma \Vdash @_a((\mathsf{match}(p, v)P)|{*}c?p \rightarrow P)$$

(RED-LI-SEND)

$$\Gamma \Vdash @_a\langle b@?\rangle c!v \ \rightarrow\ \Gamma \Vdash @_b c!v$$

Figure 4.3: Reduction semantics I

Note that the only inter-site communication in an implementation will be for the **migrate to** reduction (RED-MIGRATE), in which the body of the migrating agent a must be sent from its current site to site s — causing the location context to be updated to $\Gamma \oplus a \mapsto s$, defined as follows:

$$(\Gamma_1,\ a : \mathtt{Agent}^Z@s',\ \Gamma_2) \oplus a \mapsto s \ \stackrel{\text{def}}{=}\ \Gamma_1,\ a : \mathtt{Agent}^Z@s,\ \Gamma_2$$
$$\Gamma \oplus a \mapsto s \ \stackrel{\text{def}}{=}\ \Gamma \quad a \notin \mathsf{dom}(\Gamma)$$

Reduction is closed under structural congruence, parallel composition, and **new** binding, as specified by the rules below.

(RED-CONG)
$$\frac{LP \equiv LP' \quad \Gamma \Vdash LP \to \Gamma' \Vdash LQ \quad LQ \equiv LQ'}{\Gamma \Vdash LP' \ \to\ \Gamma' \Vdash LQ'}$$

(RED-PRL)
$$\frac{\Gamma \Vdash LP \to \Gamma' \Vdash LR}{\Gamma \Vdash LP \mid LQ \ \to\ \Gamma' \Vdash LR \mid LQ}$$

(RED-NEWAGENT)
$$\frac{\Gamma,\ x : \mathtt{Agent}^Z@s \Vdash LP \ \to\ \Gamma',\ x : \mathtt{Agent}^Z@s' \Vdash LQ}{\Gamma \Vdash \mathbf{new}\ x : \mathtt{Agent}^Z@s\ \mathbf{in}\ LP \ \to\ \Gamma' \Vdash \mathbf{new}\ x : \mathtt{Agent}^Z@s'\ \mathbf{in}\ LQ}$$

(RED-NEWCHANNEL)
$$\frac{\Gamma,\ x : \hat{\ }^I T \Vdash LP \ \to\ \Gamma',\ x : \hat{\ }^I T \Vdash LQ}{\Gamma \Vdash \mathbf{new}\ x : \hat{\ }^I T\ \mathbf{in}\ LP \ \to\ \Gamma' \Vdash \mathbf{new}\ x : \hat{\ }^I T\ \mathbf{in}\ LQ}$$

Figure 4.4: Reduction semantics II

Here, following [AP94], we present a *fine-grain* semantics, in that inter-agent communication happens in two steps: messaging (RED-LI-SEND or RED-IFLOCAL-TRUE) and synchronisation (RED-COMM). Alternatively, we could define a *coarse-grain* semantics (as in eg. [Sew98]) where inter-agent communication happens in single reduction step. Rule (RED-LI-SEND) could then be replaced by the following:

$$\Gamma \Vdash @_b\langle a@?\rangle c\mathbf{!}v \mid @_a c\mathbf{?}p \to P \qquad \to \qquad \Gamma \Vdash @_a \mathsf{match}(p, v) P$$

The implementation of the coarse grain semantics requires some amount of global synchronisation, and a handshaking procedure between the input and the output processes (see also the discussion in Section 8.1 on page 156). In the fine grain semantics, however, only a single message needs to be sent via the network; this style of semantics can therefore be implemented more efficiently. More crucial to the correctness proof, the fine-grain message delivery step of LD primitives, eg. $\langle b@s\rangle c!v$, is a deterministic reduction (see Definition 6.3.2 on page 101). This allows the temporary immobility technique to be used for ensuring that such a message will be safely delivered (see an example in Section 6.4).

4.3 Labelled Transition Semantics

The reduction semantics describes only the internal behaviour of processes — for compositional reasoning we need also a labelled transition semantics, expressing how processes can interact with their environment. The transition relations have the following forms, for basic and located processes:

$$\Gamma \Vdash_a \; P \xrightarrow[\Delta]{\alpha} LP \qquad\qquad \Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$$

Here the *unlocated labels* α are of the following forms:

τ	internal computation
$\mathsf{migrate\ to}\ s$	migrate to the site s
$c!v$	send value v along channel c
$c?v$	receive value v from channel c

The *located labels* β are of the form τ or $@_a\alpha$ for $\alpha \neq \tau$. Private names (together with their types, which may be annotated with an agent's current site) may be exchanged in communication, and are made explicit in the transition relation by the extruded context Δ. We assume that names in the extruded contexts are fresh and therefore $\mathsf{dom}(\Delta)\cap\mathsf{dom}(\Gamma) = \emptyset$. The example below shows an agent b announcing its presence to another agent a outside its scope. This involves two steps: first, the high-level output $@_b\langle a@?\rangle c!b$ delivers the message $c!b$ to a (a τ step); then such a message produces an output action $@_a c!b$, and extrudes the scope of b together with the annotated type $\mathtt{Agent}^{\mathtt{m}}@s$ to the environment.

$$\begin{array}{ll} \Gamma \Vdash \mathbf{new}\ b : \mathtt{Agent}^{\mathtt{m}}@s\ \mathbf{in}\ (@_b\langle a@?\rangle c!b \mid @_bP) & \\ \xrightarrow{\quad\tau\quad} & \mathbf{new}\ b : \mathtt{Agent}^{\mathtt{m}}@s\ \mathbf{in}\ (@_a\langle c\rangle b! \mid @_bP) \\ \xrightarrow[b:\mathtt{Agent}^{\mathtt{m}}@s]{@_a c!b} & @_bP \end{array}$$

Adding $\mathsf{migrate\ to}\ s$ to the standard input/output and τ labels is an important design choice, made for the following reasons.

- Imagine a process LP in a program context. If an agent a in LP migrates, the location context is consequently updated with a associated to its new site. This change of location context has an effect on both LP and its environment, since it can alter their execution paths (especially those involving LD communication with a). Migration of an agent must therefore be thought of as a form of interaction with the environment.

- We observe, in the reduction rules, that the location context in the configuration *after* the transition can only be modified by migration of an agent (see (RED-MIGRATE)). Including this migrating action allows the location context on the right hand side to be omitted.

Execution of other agent primitives (ie. **`create`** and **`iflocal`**) is regarded as internal computation, since it does not have an immediate effect on program contexts. In the case of **`create`**, the newly created agent remains unknown to the environment unless its name is extruded by an output action. Again we choose a fine-grain semantics where inter-agent communication involves two steps of transition. Execution of **`iflocal`** (if successful) and LI-output does not therefore produce an output action immediately.

Basic processes The labelled transition semantics for basic process is a transition relation from *basic process configurations* to located processes. A basic process configuration is an annotated pair $\Gamma \Vdash_a P$ of located type context Γ and basic process P, located at agent a. We are concerned only with configurations for well-typed programs, and define a transition relation $\xrightarrow[\Delta]{\alpha}$ as the smallest relation from $\{\Gamma \Vdash_a P \mid \vdash_{\mathsf{L}} \Gamma,\ \Gamma \vdash_a P \text{ and } \vdash \Gamma, \Delta\}$ to $\{\Gamma \Vdash LP \mid LP \in \mathrm{n}\pi_{\mathsf{LD,LI}}\}$, closed under the rules given in Figures 4.5-4.7.

The rules relating to input and output actions (given in Figure 4.6) are akin to those of a standard asynchronous π-calculus labelled transition semantics, extended with a type system. An output produces an output action in the obvious way (LTS-L-OUT). The rule (LTS-L-OPEN) ensures that if any private name x (created by a **`new`**-binding) is an argument of output action then the scope of x must be extruded along with the output label. Since only extensible names (ie. channels or agents) can be dynamically created, (LTS-L-(REP-)IN) ensures that extruded names are always of extensible types. The rule (LTS-L-COMM) permits synchronisation of an input and an output action, provided that they are co-located. As a result of channel communication, extruded contexts are binding over the communicating parties. The symmetric version of (LTS-L-COMM) is omitted.

The rules given in Figure 4.5 can be derived from their reduction rule counterparts by removing the agent annotation from the process and placing it as part of the configuration,

(LTS-L-OUT)

$$\Gamma \Vdash_a c!v \xrightarrow{c!v} @_a\mathbf{0}$$

(LTS-L-COMM)

$$\frac{\Gamma \Vdash_a P \xrightarrow[\Delta]{c!v} LP \quad \Gamma \Vdash_a Q \xrightarrow[\Delta]{c?v} LQ}{\Gamma \Vdash_a P \mid Q \xrightarrow{\tau} \mathbf{new}\ \Delta\ \mathbf{in}\ (LP \mid LQ)}$$

(LTS-L-(REP-)IN)

$$\frac{\Gamma \vdash c \in {\hat{}}^{\mathtt{r}}T \quad \Gamma, \Delta \vdash v \in T \quad \mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v) \quad \Delta \text{ extensible.}}{\begin{array}{l}\Gamma \Vdash_a c?p \rightarrow P \xrightarrow[\Delta]{c?v} @_a\mathsf{match}(p,v)P \\ \text{and } \Gamma \Vdash_a *c?p \rightarrow P \xrightarrow[\Delta]{c?v} @_a(\mathsf{match}(p,v)P \mid *c?p \rightarrow P)\end{array}}$$

(LTS-L-OPEN)

$$\frac{\Gamma,\ x : T \Vdash_a P \xrightarrow[\Delta]{c!v} LP \quad x \in \mathsf{fv}(v) \quad x \neq c}{\Gamma \Vdash_a \mathbf{new}\ x : T\ \mathbf{in}\ P \xrightarrow[\Delta, x:T]{c!v} LP}$$

Figure 4.5: Labelled transition semantics: Basic processes — input and output

replacing $\rightarrow$ by $\xrightarrow{\tau}$ and removing located type contexts on the right hand side. The only exception is (LTS-L-MIGRATE), where instead of updating the located type context, the rule produces a 'migrate to s' label. Again the high-level calculus has the additional axiom (LTS-L-LI-SEND), which can be derived from its reduction rule counterpart in the same way.

As for the reduction relation, the labelled transition relation is closed under parallel composition and **new** binding (provided the private name is not involved in the action). The symmetric version of (LTS-L-PRL) is omitted. Unlike the reduction semantics, however, the labelled transition semantics does not rely on the structural congruence; we do not need the semantics to be closed under structural congruence on the left. Matching of the reduction and the labelled transition semantics, however, requires the closure under structural congruence on the right. This allows, for example, null processes produced by an output action to be garbage-collected, as shown below.

$$\Gamma \Vdash @_a(c!v | c?p \rightarrow P) \quad \xrightarrow{\tau} \quad @_a\mathbf{0} \mid @_a(\mathsf{match}(p,v)P) \ \equiv\ @_a(\mathsf{match}(p,v)P)$$

The closure of the labelled transition relation are specified by the rules given in Figure 4.7. Note that, since the semantics is typed, the x declared in (LTS-L-NEW) must be of channel type; this means no location annotation is needed in the premise.

(LTS-L-CREATE)

$$\frac{\Gamma \vdash a@s}{\Gamma \Vdash_a \ \texttt{\textbf{create}}^Z\ b = P\ \texttt{\textbf{in}}\ Q \ \xrightarrow{\tau}\ \texttt{\textbf{new}}\ b : \texttt{Agent}^Z@s\ \texttt{\textbf{in}}\ (@_b P \mid @_a Q)}$$

(LTS-L-MIGRATE)

$$\Gamma \Vdash_a \ \texttt{\textbf{migrate to}}\ s \rightarrow P \ \xrightarrow{\mathsf{migrate\ to}\ s}\ @_a P$$

(LTS-L-IFLOCAL-TRUE)

$$\frac{\Gamma \vdash a@s \quad \Gamma \vdash b@s}{\Gamma \Vdash_a \ \texttt{\textbf{iflocal}}\ \langle b\rangle c!v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q \ \xrightarrow{\tau}\ @_a P \mid @_b c!v}$$

(LTS-L-IFLOCAL-FALSE)

$$\frac{\Gamma \vdash a@s \quad \Gamma \vdash b@s' \quad s \neq s'}{\Gamma \Vdash_a \ \texttt{\textbf{iflocal}}\ \langle b\rangle c!v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q \ \xrightarrow{\tau}\ @_a Q}$$

(LTS-L-LET)

$$\frac{\texttt{eval}(ev)\ \text{defined}}{\Gamma \Vdash_a \ \texttt{\textbf{let}}\ p = ev\ \texttt{\textbf{in}}\ P \ \xrightarrow{\tau}\ @_a \mathsf{match}(p, \texttt{eval}(ev))\ P}$$

(LTS-L-COND-TRUE)

$$\Gamma \Vdash_a \ \texttt{\textbf{if true then}}\ P\ \texttt{\textbf{else}}\ Q \ \xrightarrow{\tau}\ @_a P$$

(LTS-L-COND-FALSE)

$$\Gamma \Vdash_a \ \texttt{\textbf{if false then}}\ P\ \texttt{\textbf{else}}\ Q \ \xrightarrow{\tau}\ @_a Q$$

(LTS-L-LI-SEND)

$$\Gamma \Vdash_a \ \langle b@?\rangle c!v \ \xrightarrow{\tau}\ @_b c!v$$

Figure 4.6: Labelled transition semantics: Basic processes — internal computation

(LTS-L-NEW)
$$\frac{\Gamma,\ x:T \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP \quad x \notin \mathsf{fv}(\alpha)}{\Gamma \Vdash_a\ \mathtt{new}\ x:T\ \mathtt{in}\ P \xrightarrow[\Delta]{\alpha} \mathtt{new}\ x:T\ \mathtt{in}\ LP}$$

(LTS-L-PRL)
$$\frac{\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP}{\Gamma \Vdash_a\ P \mid Q \xrightarrow[\Delta]{\alpha} LP \mid @_a Q}$$

(LTS-L-CONG-R)
$$\frac{\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP \quad LP \equiv LQ}{\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LQ}$$

Figure 4.7: Labelled transition semantics: Basic processes — closure

Located Processes Computation of located processes is based on computation at the basic process level. The labelled transition semantics for located process is a transition relation from configurations to located processes. Again, we are concerned only with configurations for well-typed programs, and define a transition relation $\xrightarrow[\Delta]{\beta}$ as the smallest relation from $\{\Gamma \Vdash LP \mid \ \vdash_{\mathsf{L}} \Gamma,\ \Gamma \vdash LP \text{ and } \vdash \Gamma, \Delta\}$ to $\{\Gamma \Vdash LQ \mid LQ \in \mathrm{n}\pi_{\mathsf{LD,LI}}\}$, closed under the rules given in Figures 4.8-4.9.

(LTS-LOCAL) annotates the visible action performed by a basic process, relating the two forms of labelled semantics. (LTS-NEW), (LTS-PRL) and (LTS-CONG-R) ensure labelled transitions for located processes are closed under structural congruence (though again only on the right), parallel composition and **new**-binding (provided the private name is not involved in the action). The symmetric versions of (LTS-COMM) and (LTS-PRL) are omitted.

The rules involving **new** declaration of located processes are similar to those of basic processes, although they have to additionally deal with agent declaration. Migration of a bound agent is not observable, since it has no effect on the environment. In this case, the site annotation of the migrating agent is "silently" updated (LTS-BOUND-MIG). As in (LTS-L-OPEN), a private name x can be extruded by an output action — also, if x is an agent then its site annotation must be attached. We insist that x must not be the locator of the output action. This prevents, for example, $\mathtt{new}\ x : \mathtt{Agent}^{\mathtt{m}}@s\ \mathtt{in}\ @_x c!v$ from performing an output action, even though the environment cannot have an input process located at x that may react to the output.

Early versus Late Semantics Here we have described an *early* form of labelled transition semantics, where the value involved in an input action is instantiated at the time when the action is being performed. Alternatively, the *late* form of labelled transition semantics gives

(Lts-Local)

$$\frac{\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP \quad \beta = \begin{cases} \tau & \text{if } \alpha = \tau \\ @_a\alpha & \text{otherwise} \end{cases}}{\Gamma \Vdash @_a P \xrightarrow[\Delta]{\beta} LP}$$

(Lts-Prl)

$$\frac{\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LR}{\Gamma \Vdash LP \mid LQ \xrightarrow[\Delta]{\beta} LR \mid LQ}$$

(Lts-Comm)

$$\frac{\Gamma \Vdash LP \xrightarrow[\Delta]{@_a c\,\mathtt{!}\,v} LP' \quad \Gamma \Vdash LQ \xrightarrow[\Delta]{@_a c\,\mathtt{?}\,v} LQ'}{\Gamma \Vdash LP \mid LQ \xrightarrow{\tau} \mathbf{new}\ \Delta\ \mathbf{in}\ (LP' \mid LQ')}$$

(Lts-Cong-R)

$$\frac{\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ \quad LQ \equiv LR}{\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LR}$$

(Lts-NewChannel)

$$\frac{\Gamma,\ x : \mathtt{\char94}^I T \Vdash LP \xrightarrow[\Delta]{\beta} LQ \quad x \notin \mathsf{fv}(\beta)}{\Gamma \Vdash \mathbf{new}\ x : \mathtt{\char94}^I T\ \mathbf{in}\ LP \xrightarrow[\Delta]{\beta} \mathbf{new}\ x : \mathtt{\char94}^I T\ \mathbf{in}\ LQ}$$

(Lts-NewAgent)

$$\frac{\Gamma,\ x : \mathtt{Agent}^Z @s \Vdash LP \xrightarrow[\Delta]{\beta} LQ \quad x \notin \mathsf{fv}(\beta)}{\Gamma \Vdash \mathbf{new}\ x : \mathtt{Agent}^Z @s\ \mathbf{in}\ LP \xrightarrow[\Delta]{\beta} \mathbf{new}\ x : \mathtt{Agent}^Z @s\ \mathbf{in}\ LQ}$$

Figure 4.8: Labelled transition semantics: Located processes — basic

(Lts-Bound-Mig)

$$\frac{\Gamma,\ a : \mathtt{Agent}^{\mathtt{m}} @s \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s'} LQ}{\Gamma \Vdash \mathbf{new}\ a : \mathtt{Agent}^{\mathtt{m}} @s\ \mathbf{in}\ LP \xrightarrow{\tau} \mathbf{new}\ a : \mathtt{Agent}^{\mathtt{m}} @s'\ \mathbf{in}\ LQ}$$

(Lts-OpenChannel)

$$\frac{\Gamma,\ x : \mathtt{\char94}^I T \Vdash LP \xrightarrow[\Delta]{@_a c\,\mathtt{!}\,v} LQ \quad x \in \mathsf{fv}(v) \quad x \neq c \quad x \neq a}{\Gamma \Vdash \mathbf{new}\ x : \mathtt{\char94}^I T\ \mathbf{in}\ LP \xrightarrow[\Delta, x:\mathtt{\char94}^I T]{@_a c\,\mathtt{!}\,v} LQ}$$

(Lts-OpenAgent)

$$\frac{\Gamma,\ x : \mathtt{Agent}^Z @s \Vdash LP \xrightarrow[\Delta]{@_a c\,\mathtt{!}\,v} LQ \quad x \in \mathsf{fv}(v) \quad x \neq c \quad x \neq a}{\Gamma \Vdash \mathbf{new}\ x : \mathtt{Agent}^Z @s\ \mathbf{in}\ LP \xrightarrow[\Delta, \mathtt{Agent}^Z @s]{@_a c\,\mathtt{!}\,v} LQ}$$

Figure 4.9: Labelled transition semantics: Located processes — new declaration

input actions a functional operational intuition: when inputing, a process becomes a function of the actual transmitted name. The examples below show the late transitions of input and output processes.

$$x(y).P \xrightarrow{x} (y).P \qquad\qquad \bar{x}\langle z\rangle.Q \xrightarrow{\bar{x}} \langle z\rangle.Q$$

Communication between the above processes may occur. This involves "fusing" $(y).P$ and $\langle z\rangle.Q$ to obtain $\{y/z\}P \mid Q$.

It is a matter of taste which semantics to adopt. Parrow [Par00] claims that the proof systems and decision procedures of late semantics are slightly more efficient. On the other hand, it could be argued that the early semantics follows more closely an operational intuition since, after all, an agent performs an input action only when it actually receives a particular value; it is on this line of reasoning that we adopt the early semantics. Other styles of semantics include *open semantics* [San96b] and *symbolic transitions* [Lin94]. We omit their definitions; reader may refer to [Par00] for further details.

Alternative choices In Nomadic π-calculi, processes located at the same agent may disperse throughout a located process. This flexibility of syntax allows processes to make a transition without any need to be rearranged (by means of structural congruence) so that it fits a particular form (cf. (RED-COMM) and Theorem 4.5.2). We may imagine a more restricted treatment of agent distribution where all processes are located at the same agent must be grouped together (eg. the LTS presented in [CG98]). This requires the rules with across-agent effects need to be treated with care. However, we argue that our treatment of distribution is more suited to compositional reasoning since it allows equivalences between fragments of located processes to be defined. In Lemma 6.4.3, for example, to show that the following process is temporarily immobile, we need to analyse its possible transitions, without considering other parts of the daemon D (especially the daemon lock).

$$\begin{array}{l}\mathbf{new}\ \Omega_{aux}\ \mathbf{in}\ @_D(Daemon|\prod_i \mathsf{mesgReq}\{T_i\}\ [a\ c_i\ v_i]) \\ \quad \mid @_a([\![P]\!]_a\,|\mathtt{currentloc!}\,s|\mathtt{Deliverer})\end{array}$$

There are alternative design choices for the labelled transition semantics; the author has experimented with the following.

1. $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} \Gamma' \Vdash LQ$: The LTS is in the same form as the reduction semantics. In this case the located label β can be τ, $@_a c\mathbf{?}v$ and $@_a c\mathbf{!}v$.

   Potential problem: since the transition relation is now between configurations, operational equivalences should now compare the behaviour exhibited by configurations.

This allows behavioural comparison of systems of agents, even though the location contexts of such systems are different. For example, executing a location-independent output $\langle b@?\rangle c\,!\,v$ as part of an agent a should deliver $c\,!\,v$ to b, regardless of where a, b are. More formally, provided that $\Gamma_i \vdash @_a\langle b@?\rangle c\,!\,v$, for $i = 1, 2$, we may derive the following.

$$\Gamma_1 \Vdash @_a\langle b@?\rangle c\,!\,v \quad \dot{\sim} \quad \Gamma_2 \Vdash @_a\langle b@?\rangle c\,!\,v$$

where $\dot{\sim}$ is a standard strong bisimulation relation. This kind of property might be desirable in many cases, but the meta-theory for such operational equivalences is likely to be complex.

2. $\Gamma \Vdash_a \; P \xrightarrow[\Delta]{\alpha} Q$: Our labelled transition relation for basic process is from (basic process) configurations to located processes, since executing agent creation and inter-agent communication primitives produces processes which are located at other agents. To obtain this new form for basic process transition, we would need to extend the syntax of α to signify the effects caused by such primitives. For example, we may extend the syntax of basic process labels as follows.

$$\beta \quad ::= \quad \ldots \mid \mathbf{create}^Z\; b = P$$

(Lts-L-Create) could then be replaced by the following rules.

$$\text{(Lts-L-CreateLabel)}\quad \Gamma \Vdash_a \; \mathbf{create}^Z\; b = P \;\mathbf{in}\; Q \xrightarrow{\mathbf{create}^Z\; b=P} Q$$

$$\text{(Lts-Creating)}\quad \frac{\Gamma \vdash a@s \quad \Gamma \Vdash_a \; Q \xrightarrow{\mathbf{create}^Z\; b=P} R}{\Gamma \Vdash @_a Q \xrightarrow{\tau} \mathbf{new}\; b : \mathtt{Agent}^Z@s \;\mathbf{in}\; @_b P \mid @_a R}$$

This alternative poses many difficulties. First one is that of scoping: after the execution, the scope of b in $\mathbf{create}^Z\; b = P \;\mathbf{in}\; Q$ must be over $@_a Q \mid @_b P$. This means that a transition which undergoes a create action must not be closed under parallel composition. More crucially, based on the work on higher-order process calculi of Sangiorgi [San96a], a labelled transition semantics in which processes are included in the syntax of labels are likely to give rise to operational equivalences with undesirable discriminative power.

4.4 Runtime Errors

The operational semantics do not define the situations in which executing a located process leads to runtime errors. To ensure that the type system prevents such runtime errors, we identify the following situations in which errors may occur.

- Mismatching: a located process LP contains an output $c\texttt{!}v$ and an input $c\texttt{?}p \rightarrow P$ in the same agent, but $\mathsf{match}(p, v)$ is undefined.
- Incapability: a located process LP, well-typed w.r.t. Γ, attempts to migrate a static agent, use an input-only channel for output, or use an output-only channel for input.
- Invalid expression: evaluating the value in a well-typed **let** expression or conditional yields an undefined result.

The type system does not always guarantee the absence of such errors, though. The examples below show the situations in which mismatching and invalid expression occur when executing well-typed programs.

$$
\begin{array}{ll}
s : \texttt{Site},\ a : \texttt{Agent}^{\texttt{m}}@s,\ c : \texttt{\^{}}^{\texttt{rw}}[[][]],\ x : [[][]] \Vdash_a\ (c\texttt{!}x \mid c\texttt{?}[y\ z] \rightarrow \mathbf{0}) & \not\xrightarrow{\tau} \\
s : \texttt{Site},\ a : \texttt{Agent}^{\texttt{m}}@s,\ x : \texttt{Bool} \Vdash_a\ \textbf{if}\ x\ \textbf{then}\ P\ \textbf{else}\ Q & \not\xrightarrow{\tau}
\end{array}
$$

In the first situation, communication between the input and output processes cannot occur since $\mathsf{match}([y\ z], x)$ is undefined. This type of situation can be avoided by working with programs which are well-typed w.r.t. ground type contexts. In the second situation, the process contains a name of base type, which cannot be evaluated. To avoid the above situations, we define *closed* type contexts as follows.

Definition 4.4.1 (Closed Type Context)
A type context Γ is closed if it is ground and $\mathsf{fv}(\Gamma) \cap \mathcal{TV} = \emptyset$ and, for all $x \in \mathsf{dom}(\Gamma)$, $\Gamma \vdash x \in T$ implies $T \notin \mathcal{T}$.

It is easy to show that each name declared in a closed type context is either a site, an agent, or a channel. The following three lemmas ensure that no situation discussed above may occur in processes well-typed w.r.t. closed type contexts.

Lemma 4.4.1 (Runtime safety: Channels)
Given that Γ is a closed type context, and $(\Gamma, \Delta)(c) = \texttt{\^{}}^I T$, we have:

1. if $\Gamma \vdash \textbf{new}\ \Delta\ \textbf{in}\ (@_a c\texttt{!}v \mid LP)$ then $I \leq \texttt{w}$;
2. if $\Gamma \vdash \textbf{new}\ \Delta\ \textbf{in}\ (@_a c\texttt{?}p \rightarrow P \mid LP)$ then $I \leq \texttt{r}$;

3. if $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a \mathtt{*}c\mathtt{?}p \rightarrow P \mid LP)$ then $I \leq \mathtt{r}$;
4. if $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(c\mathtt{!}v | c\mathtt{?}p \rightarrow P) \mid LP)$ then $\mathsf{match}(p, v)$ is defined; and
5. if $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(c\mathtt{!}v | \mathtt{*}c\mathtt{?}p \rightarrow P) \mid LP)$ then $\mathsf{match}(p, v)$ is defined.

Proof: The first three properties are straightforward examination of the typing rules. The last two rely on Lemma 3.8.3. ■

Lemma 4.4.2 (Runtime safety: Expressions)
Given that Γ is a closed type context, we have:

1. if $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q) \mid LP)$ then $v \in \{\mathbf{true}, \mathbf{false}\}$;
2. if $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathbf{let}\ p = ev\ \mathbf{then}\ P) \mid LP)$ then $\mathtt{eval}(ev)$ and $\mathsf{match}(p, \mathtt{eval}(ev))$ are defined.

Proof: The first property is easy to establish. The second property is proved by an induction on the syntax of ev. The proof relies on the fact that each basic function $f \in \mathcal{F}$ is a total function (for $\mathtt{eval}(ev)$ being defined), and Lemma 3.8.3 (for $\mathsf{match}(p, \mathtt{eval}(ev))$ being defined). ■

Lemma 4.4.3 (Immobility implies no migration)
Given that $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ LP$, if $(\Gamma, \Delta)(a) = \mathtt{Agent}^{\mathtt{s}}@s$ then there exists *no* LQ and s' such that

1. $a \in \mathsf{dom}(\Gamma)$ and $\Gamma \Vdash \mathbf{new}\ \Delta\ \mathbf{in}\ LP \xrightarrow{@_a \mathsf{migrate\ to}\ s'} LQ$;
2. $a \in \mathsf{dom}(\Delta)$ and $\Gamma \Vdash \mathbf{new}\ \Delta\ \mathbf{in}\ LP \quad \xrightarrow{\tau} \quad \mathbf{new}\ (\Delta \oplus a \mapsto s')\ \mathbf{in}\ LQ$.

Proof: We may prove by an induction on transition derivation that if $a \in \mathsf{dom}(\Gamma)$ and $\Gamma, \Delta \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s'} LQ$ then $\Gamma, \Delta \vdash a \in \mathtt{Agent}^{\mathtt{m}}$. This leads to a contradiction with the premise $(\Gamma, \Delta)(a) = \mathtt{Agent}^{\mathtt{s}}@s$. The assumption cannot be true, and hence the lemma.

A similar argument applies for the case where $a \in \mathsf{dom}(\Delta)$. ■

The intuition about runtime errors could alternatively be formalised by defining an error predicate on terms, as in eg. [PS96, RH98], with $P \overset{err}{\rightarrow}$ meaning that P may immediately produce a runtime error. Below we sketch some example rules of such as error predicate on *untyped* configuration, written $L \Vdash LP$, where LP may be ill-typed, and L ranges over location contexts, mapping agents to their current locations. Each agent and channel name must be tagged with its capability, so that violations of capability can be checked in these rules. The rules given are for detecting mismatching of the value sent to and expected from a channel (MISMATCH), violation of type capabilities (INCAPABLE-MIG), and propagation of errors through parallel composition and new binding (PROPAGATE).

$$\frac{\mathsf{match}(p,v) \text{ undefined} \quad \text{or} \quad I \not\leq \mathtt{w} \quad \text{or} \quad I' \not\leq \mathtt{r}}{L \Vdash_{a} z \;\; c^{I}\,\mathbf{!}\,v \mid c^{I'}\,\mathbf{?}p \rightarrow P \;\xrightarrow{err}} \qquad \text{(MISMATCH)}$$

$$\frac{Z = \mathtt{s}}{L \Vdash_{a} z \;\; \mathbf{migrate\ to}\ s \rightarrow P \;\xrightarrow{err}} \qquad \text{(INCAPABLE-MIG)}$$

$$\frac{L \Vdash LP \xrightarrow{err}}{L \Vdash LP \mid LQ \xrightarrow{err}} \qquad \frac{L \Vdash LP \xrightarrow{err}}{L/\mathsf{dom}(\Delta) \Vdash \mathbf{new}\ \Delta\ \mathbf{in}\ LP \xrightarrow{err}} \qquad \text{(PROPAGATE)}$$

Given the above error predicate, the runtime safety result could be stated as follows.

$$\text{If } \Gamma \vdash LP \text{ and } \Gamma \vdash L \text{ then } L \Vdash LP \overset{err}{\not\longrightarrow}$$

where $\Gamma \vdash L$ ensures that names in the domain and the range of L are of the correct types. This runtime safety result is clearly more precise than that given in this subsection. The cost is that the error predicate has to be defined for every syntactic constructs and possible scenarios of runtime error. We believe an error predicate can easily be given for Nomadic π-calculi.

4.5 Properties of the Semantics

We prove a number of properties of our semantics. The first three results, together with those in Section 4.4 which deal with absence of runtime errors, constitute soundness theorems of the Nomadic π-calculi. The results given here relate the type system, the structural congruence, and the two semantics. A subject-reduction theorem (Theorem 4.5.1) states that well-typed processes remains well-typed after a successful transition step. Theorem 4.5.2 shows that process transitions are preserved by structural congruence. This theorem simplifies the task of exhaustively enumerating transitions, since we only need to check one form of a process and be sure that no more transition is possible in other structurally congruent forms. Finally, Theorem 4.5.3 shows that the two semantics coincide in the absence of input or output actions.

Theorem 4.5.1 (Subject reduction)
Given a closed located type context Γ,

1. if $\Gamma \Vdash_a \; P \xrightarrow[\Delta]{\alpha} LP$ then $\Gamma, \Delta \vdash LP$; and

2. if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\Gamma, \Delta \vdash LQ$.

Proof: An induction on the derivations of $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ and $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$. Γ needs to be closed so that, matching a pattern with a value of the same type always yields a type-preserving substitution, whenever the transition involves matching occurs (eg. that derived by LTS-L-LET or LTS-L-(REP-)IN). See full details in Appendix B.3 on page 203. ■

Theorem 4.5.2 (Cong-L absorption)
Given a closed located type context Γ,

1. if $P \equiv Q$ and $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ then $\Gamma \Vdash_a Q \xrightarrow[\Delta]{\alpha} LP$; and
2. if $LP \equiv LR$ and $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\Gamma \Vdash LR \xrightarrow[\Delta]{\beta} LQ$.

Proof: This is proved by an induction on the derivation of $P \equiv Q$ and $LP \equiv LQ$. Much of this proof built on that shown in [Sew00] — only the rules (STR-LOCATE) and (STR-DISTR) are specific to our setting.

Full details can be found in Appendix B.4 on page 206. ■

Theorem 4.5.3 (Labelled and reduction semantics matching)
Given a closed located type context Γ, and a located process LP,

$$\Gamma \Vdash LP \rightarrow \Gamma' \Vdash LQ \quad \text{IFF} \quad \begin{cases} \Gamma \Vdash LP \xrightarrow{\tau} LQ & \Gamma' = \Gamma \text{ or} \\ \Gamma \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s} LQ & \Gamma' = \Gamma \oplus a \mapsto s \end{cases}$$

Proof: We need to show this in two parts: that a reduction implies a silent transition or a migrate action, and vice versa. Each of the two parts is shown by an induction on reduction/transition derivations. The case where the silent transition of LP is derived by (LTS-COMM) needs the following lemma, which can easily be proved by an induction on transition derivations (details omitted).

Lemma 4.5.4

- If $\Gamma \Vdash LP \xrightarrow[\Xi]{@_a c!v} LQ$ then $LP \equiv \mathbf{new}\ \Delta, \Xi\ \mathbf{in}\ (@_a c!v \mid LP')$ for some Δ and LP'. Moreover, $LQ \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ LP'$.
- If $\Gamma \Vdash LP \xrightarrow[\Xi]{@_a c?v} LQ$ then, for some Δ, p and LP', Q, with $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Xi) = \emptyset$, either:
  - $LP \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a c?p \rightarrow Q \mid LP')$ and $LQ \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathsf{match}(p, v)Q) \mid LP')$; or
  - $LP \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a {*}c?p \rightarrow Q \mid LP')$ and $LQ \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathsf{match}(p, v)Q) \mid @_a {*}c?p \rightarrow Q \mid LP')$.

- If $\Gamma \Vdash LP \xrightarrow{@_a\mathsf{migrate\ to}\ s} LQ$ then

$$LP \equiv \texttt{new}\ \Delta\ \texttt{in}\ (@_a\texttt{migrate to}\ s \rightarrow P \mid LP')$$

for some Δ and LP', P. Moreover, $LQ \equiv \texttt{new}\ \Delta\ \texttt{in}\ (@_a P \mid LP')$.

As in Theorem 4.5.1, Γ needs to be closed so that, matching a patterns with a value of the same type always yields a type-preserving substitution, whenever the transition involves matching occurs.

Full details may be found in Appendix B.5 on page 208. ■

The remaining lemmas are required for proving meta-theoretic results and the correctness of distributed infrastructures. Lemma 4.5.5 allows strengthening and weakening of the located type context Γ in transition relations by a Θ, provided that the bindings in Θ do not clash with Γ or with the extruded names. The side-condition of this lemma is delicate — the following examples show why the names in Θ must neither occur in the label nor the extruded context.

$$\begin{array}{lcl} \Gamma,\ X \Vdash_a\ c\texttt{?}\,\{Y\}\,p \rightarrow P & \xrightarrow{c\texttt{?}\{X\}v} & @_a\mathsf{match}(\{Y\}\,p, \{X\}\,v)P \\ \Gamma,\ s:\texttt{Site} \Vdash_a\ x\texttt{?}p \rightarrow P & \xrightarrow[a:\texttt{Agent}^{\texttt{m}}@s]{x\texttt{?}a} & @_a\mathsf{match}(\{Y\}\,p, \{X\}\,v)P \end{array}$$

We may observe that in such cases, the names X and s are necessary for both transitions, and therefore cannot be strengthened even though they are not used in $c\texttt{?}\,\{Y\}\,p \rightarrow P$ or $x\texttt{?}p \rightarrow P$.

Lemma 4.5.6 shows that extensible name arguments in an input action can be considered a part of the extruded context or of the main context. Lemma 4.5.7 shows that labelled transitions are preserved by injective substitutions. Again this lemma does not hold for an arbitrary name substitution, since the semantics is typed and applying a non-injective substitution to a well-typed configuration may result in an ill-typed one. For example, let Γ be $s:\texttt{Site}, s':\texttt{Site}$ and σ be $\{s/s'\}$; clearly $\sigma\Gamma$ is not a well-formed type context. The proofs of these lemmas use induction on the transition derivations; they can be found in Appendix B.2 on page 192.

Lemma 4.5.5 (Weakening and strengthening transition)
Given that $\vdash_{\mathsf{L}} \Gamma, \Theta$,

1. $\Gamma \vdash_a P$ and $\mathsf{dom}(\Theta) \cap (\mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)) = \emptyset$ implies

$$\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP \ \Leftrightarrow\ (\Gamma, \Theta) \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP$$

2. $\Gamma \vdash LP$ and $\mathsf{dom}(\Theta) \cap (\mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)) = \emptyset$ implies

$$\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ \ \Leftrightarrow\ (\Gamma, \Theta) \Vdash LP \xrightarrow[\Delta]{\beta} LQ$$

Lemma 4.5.6 (Shifting: input transitions)
Given that Θ is an extensible context with $a, c \notin \mathsf{dom}(\Theta)$ and $\mathsf{dom}(\Theta) \subseteq \mathsf{fv}(v)$, we have:

1. $\Gamma \vdash_a P$ and $\vdash_{\mathsf{L}} \Gamma, \Theta$ implies $\Gamma \Vdash_a P \xrightarrow[\Delta,\Theta]{c\boldsymbol{?}v} LP \Leftrightarrow (\Gamma, \Theta) \Vdash_a P \xrightarrow[\Delta]{c\boldsymbol{?}v} LP$; and
2. $\Gamma \vdash LP$ and $\vdash_{\mathsf{L}} \Gamma, \Theta$ implies $\Gamma \Vdash LP \xrightarrow[\Delta,\Theta]{@_a c\boldsymbol{?}v} LQ \Leftrightarrow (\Gamma, \Theta) \Vdash LP \xrightarrow[\Delta]{@_a c\boldsymbol{?}v} LQ$.

Lemma 4.5.7 (Injection preserves transition)
Given that $\rho : \mathsf{dom}(\Gamma) \to \mathcal{X}$ is injective and $\sigma : \mathsf{dom}(\Delta) \to \mathcal{X}$ is injective and $\mathsf{range}(\rho) \cap \mathsf{range}(\sigma) = \emptyset$, we have:

1. if $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ then $\rho\Gamma \Vdash_{\rho a} \rho P \xrightarrow[\sigma\Delta]{(\rho+\sigma)\alpha} (\rho + \sigma)LP$; and
2. if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\rho\Gamma \Vdash \rho LP \xrightarrow[\sigma\Delta]{(\rho+\sigma)\beta} (\rho + \sigma)LQ$.

Chapter 5

Operational Equivalences

The next two chapters form the second part of this thesis, in which we investigate semantic and proof techniques required for stating and proving the correctness of distributed infrastructures. We are expressing distributed infrastructure algorithms as encodings from a high-level calculus to its low-level fragment, so the behaviour of a source program and its encoding can be compared directly with some notion of *operational equivalence* — our main theorem will be roughly of the form

$$\forall LP \,.\, LP \simeq \mathcal{C}\,[\![LP]\!] \tag{‡}$$

where LP ranges over well-typed programs of the high-level calculus (LP may use LI communication whereas $\mathcal{C}\,[\![LP]\!]$ will not). Now, what equivalence $\simeq$ should we take? The stronger it is, the more confidence we gain that the encoding is correct. At first glance, one might take some form of weak bisimulation since (modulo divergence) it is finer than most notions of testing [dH84, Sew97] and is easier to work with. However, as in Nestmann's work on choice encodings [NP96], Equation ‡ would not hold, as the encoding $\mathcal{C}\,[\![P]\!]$ involves *partial commitment* of some nondeterministic choices (see more discussion in Section 5.4.2). We therefore take $\simeq$ to be *coupled simulation* [PS92], a slightly coarser relation, adapted to our calculi.

To prove Equation ‡, however, we need compositional techniques, allowing separate parts of the protocols to be treated separately. In particular, we need operational congruences (both equivalences and preorders) that are preserved by program contexts involving parallel composition and new-binding. In Nomadic π the behaviour of LD communications depends on the relative location of agents: if a and b are at the same site then the LD message $@_b\langle a@s\rangle c!v$ reduces to (and in fact is weakly equivalent to) the local output $@_a c!v$, whereas if they are at different sites then the LD message is weakly equivalent to $\mathbf{0}$. A parallel context, eg. $[.]|@_a\mathtt{\textbf{migrate to}}\ s \rightarrow \mathbf{0}$, can migrate the agent a, so to obtain a congruence we need refined equivalences, taking into account the possibility of such changes of agent location caused by the environment.

As in Chapter 4, we are working with *typed* operational equivalences, where the equivalences are defined exclusively over well-typed terms. The advantage of this is that properties involving such equivalences are more precise (eg. in Lemma 5.3.5, we can never strengthen or weaken an equivalence unless the located type context involved is extensible). Furthermore, types also play an important role in many definitions (eg. the definition of translocating equivalences).

This chapter is organised as follows. Section 5.1 gives some background of standard operational equivalences in process calculi, in particular, bisimulation. This standard definition is adapted to Nomadic π-calculi, adding located type contexts and migrate actions, in Section 5.2. Section 5.3 explores a number of ways in which the definition of standard bisimulation can be extended so that it takes into account movements of agents. We are looking for an operation equivalence which is a congruence for Nomadic π-calculi; we show, by using counter-examples, why these attemps are not suffice. We introduce *translocating* bisimulations, and show that they are indeed preserved by parallel composition and **new**-binding; translocating bisimulations (and other relations) will occur extensively in the thesis later on. Section 5.4 concludes this chapter by giving two other relations which are essential for our proof of correctness: an expansion adapted with translocation and a coupled simulation.

5.1 Background

Operational relations (both equivalences and preorders) provide formal ways in which the behaviour of processes can be compared, abstracting away the algebraic structure of process terms. They play an important role in the semantic theory of process calculi, allowing verification of processes by comparing them to some specifications (themselves written as processes), proofs of the correctness of encodings, and comparison between expressiveness of calculi. It is essential that such equivalences possess a reasonable discriminating power on process behaviour as well as some convenient proof techniques. There exists a large number of existing notions of equivalences in the process calculi literature, differing eg. in their treatment of internal computation, termination and divergence. Detailed analysis of these notions can be found in the surveys of van Glabbeek [vG90, vG93].

Here we shall concentrate on operational equivalences based on *simulation* relations and *bisimilarity*. Bisimilarity is an equality based on operational behaviour, introduced into computer science by Park [Par81] and developed by Milner in his theory of CCS [Mil89]. The operational behaviour of processes can be based on the labelled transition semantics or the reduction semantics equipped with some notion of *barbs*. This gives rise to two major kinds of operational equivalences: *bisimulation* and *barbed bisimulation*. Bisimulation is based on labelled transitions: a bisimulation relation is a relation between processes which is closed under transition relation. In the CCS, a binary relation $\mathcal{R}$ is a strong bisimulation if it is symmetric and, for any $P\mathcal{R}Q$, the following holds.

$$P \xrightarrow{\alpha} P' \text{ implies there exists } Q' \text{ such that } Q \xrightarrow{\alpha} Q' \text{ and } P'\mathcal{R}Q'$$

where α ranges over CCS actions $\{\tau, x, \bar{x}, \ldots\}$. Two processes are bisimilar if they are related by some bisimulation relation $\mathcal{R}$. Barbed bisimulation [MS92], on the other hand, requires only the reductions and immediate offers of communication to be matched. It can be obtained by replacing $\xrightarrow{\alpha}$ with a reduction relation $\rightarrow$, and adding the following condition which ensures matching of all possible barbs:

- $P \downarrow_x$ if and only if $Q \downarrow_x$.

For each process, its set of barbs indicates the possible interactions with its environment: informally $P \downarrow_x$ means that P can do an input or output on channel x (depending on whether x is a name or a co-name).

The advantage of bisimulation equivalences over other operational equivalences such as trace equivalence [Hoa85] or testing equivalence [dH84] is the *co-inductive* style of their definition: two processes are bisimilar if assuming that they are leads to no contradiction. Furthermore, only one computation step needs to be matched at a time (in contrast to analysing all possible paths required by the other quoted equivalence).

To simplify proofs of operational equivalences, it is often required that bisimulation equivalences are preserved by parallel composition, name binding, and (in some cases) other prefixes. This enables compositional reasoning, allowing decomposition of verification tasks and proof reuse. An operational equivalence which satisfies the above requirement is said to be a *congruence*. This does not hold for all operational equivalences. For example, the standard strong bisimulation of the π calculus is not a congruence since it is not preserved by input prefixes (see [MPW92], part II p. 14). Nevertheless, congruence properties remain important goals in defining operational equivalences.

Bisimulation equivalence and barbed bisimulation are different styles of definition, each with its merits. The definition of barbed bisimulation does not depend on an LTS, and so may be readily given for novel calculi which are equipped with reduction semantics (as is the case for the Distributed Join [FG96] and the Ambient calculus [GC99]). Nevertheless we shall be working with bisimulations (and operational relations based on the labelled transition semantics) in this thesis for two reasons. Firstly, barbed equivalence is not suitable for compositional reasoning, since barbs ignore some crucial information, eg. values exchanged in channel communication. Definitions of barbed congruence generally involve quantification over all process contexts; proving barbed congruence relation between processes can therefore be difficult. The second reason is specific to the problem of verifying infrastructure: the labelled transition semantics is required for identifying temporarily immobile processes (see Section 6.4), and for proving their key property. Using the reduction relation and the barbed congruence for dealing with temporary immobility would be awkward.

For the π-calculi, taking into account the treatment of names exchanged in input or output action leads to other variants of bisimulation equivalences. If the value involved in an input action is instantiated at the time when the action is being performed (as in the early semantics in § 4.3 on page 55) then adapting the above definition leads to an *early* bisimulation. In a *late* bisimulation, however, such an instantiation is delayed in such a way that, when a pair of bisimilar processes perform a matching input transition, it becomes a pair of processes that are bisimilar with respect to all instantiations of the values received. More formally, this requires the above definition of bisimulation to be strengthened with the following condition.

$P \xrightarrow{\alpha} P'$ with $\alpha = c(x)$ implies there exists Q' such that $Q \xrightarrow{c(x)} Q'$ and, for all y, $P'\{x/y\}\mathcal{R}Q'\{x/y\}$

where $c(x)$ denotes an input action on a channel c, receiving a name x. It is argued [MPW92, Par00] that the early style of equivalences corresponds closer to operational intuition, whereas the late style seems to lead to more efficient verification in automated tools. Other alternative notions have been proposed, for example, an *open* bisimilarity [San96b], used in the Mobility Workbench [VM94], and an *asynchronous* bisimulation [ACS98] defined for an asynchronous π-calculus.

It is sometimes convenient to visualise the relations between processes, transitions, and operational relations. Following [Fou98], such relations can be illustrated by diagrams consisting of nodes that represent configurations or located processes, linked by labelled edges that represent relations among the nodes. By convention, solid edges stand for universally-quantified relations, whereas dotted edges stand for existentially-quantified relations. The example shown below illustrates CCS processes P, Q which are related by a strong simulation $\mathcal{S}$.

$$\begin{tikzcd}
P \arrow[r, "\alpha"] \arrow[d, dash, "\mathcal{S}"] & P' \arrow[d, dotted, dash, "\mathcal{S}"] \\
Q \arrow[r, dotted, "\alpha"] & Q'
\end{tikzcd}$$

5.2 Bisimulations

In this section, we extend the standard notion of bisimulation to Nomadic π-calculi. This involves indexing bisimulation relations by the located type contexts required by the transition relation. The definition of bisimulations for Nomadic π-calculi requires the following preliminary notions. Since, in the labelled transition semantics, the located type context in the configuration needs be updated after a migration by a free agent occurs, we write $\Gamma\beta$ for the result of *relocating* Γ by β, formally defined as follows:

$$\Gamma\beta \stackrel{\text{def}}{=} \begin{cases} \Gamma_1,\ a : \texttt{Agent}^Z@s,\ \Gamma_2 & \text{if } \beta = @_a\texttt{migrate to } s \text{ and } \Gamma = \Gamma_1,\ a : \texttt{Agent}^Z@s',\ \Gamma_2. \\ \Gamma & \text{otherwise.} \end{cases}$$

We also define a *weak transition*, denoted $\overset{\beta}{\underset{\Delta}{\Longrightarrow}}$ or $\overset{\hat{\beta}}{\underset{\Delta}{\Longrightarrow}}$, generally defined as follows.

$$\overset{\beta}{\underset{\Delta}{\Longrightarrow}} \quad \stackrel{\text{def}}{=} \quad \xrightarrow{\tau}* \overset{\beta}{\underset{\Delta}{\longrightarrow}} \xrightarrow{\tau} *$$

$$\overset{\hat{\beta}}{\underset{\Delta}{\Longrightarrow}} \quad \stackrel{\text{def}}{=} \quad \begin{cases} \xrightarrow{\tau}* & \beta = \tau \\ \xrightarrow{\tau}* \overset{\beta}{\underset{\Delta}{\longrightarrow}} \xrightarrow{\tau}* & \text{otherwise} \end{cases}$$

In the context of the Nomadic π-calculi semantics, we define $\Gamma \Vdash LP \overset{\beta}{\underset{\Delta}{\Longrightarrow}} LQ$ if there exists $LP_1, \ldots, LP_n$ and $LQ_1, \ldots, LQ_k$ with $LP = LP_1$ such that:

- $\Gamma \Vdash LP_i \xrightarrow{\tau} LP_{i+1}$, for $i = 1 \ldots n-1$,
- $\Gamma \Vdash LP_n \overset{\beta}{\underset{\Delta}{\longrightarrow}} LQ_1$,
- $\Gamma \Vdash LQ_j \xrightarrow{\tau} LQ_{j+1}$, for $j = 1 \ldots k-1$, and
- $LQ_k = LQ$.

The definition of $\Gamma \Vdash LP \overset{\hat{\beta}}{\underset{\Delta}{\Longrightarrow}} LQ$ is similar to above except that if $\beta = \tau$ then the second clause is replaced by $LP_n = LQ_1$. We also define $\xrightarrow{\hat{\tau}}$ to be $\xrightarrow{\tau} \cup Id$, with Id being the identity relation. The formal definition of strong and weak simulations for Nomadic π-calculi can now be given as follows.

Definition 5.2.1 (Simulations for Located Processes)

1. A binary relation $\mathcal{S}$, indexed by closed located type contexts, and over terms of n$\pi_{\mathsf{LD,LI}}$, is a *strong simulation* if, for all well-formed Γ, $(LP, LQ) \in \mathcal{S}_\Gamma$ implies:

   - $\vdash_{\mathsf{L}} \Gamma$, $\Gamma \vdash LP$ and $\Gamma \vdash LQ$; and

   - if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ then there exists LQ' such that $\Gamma \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ and $(LP', LQ') \in \mathcal{S}_{\Gamma\beta,\Delta}$.

   $\mathcal{S}$ is called a *strong bisimulation* if all of its indexed relations are symmetric. Two located processes LP and LQ are strongly bisimilar with respect to Γ, denoted $LP \mathrel{\dot{\sim}_\Gamma} LQ$, if there exists a strong bisimulation $\mathcal{S}$ with $LP \mathcal{S}_\Gamma LQ$.

2. Replacing $\Gamma \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ in the previous item of this definition with $\Gamma \Vdash LQ \overset{\hat{\beta}}{\underset{\Delta}{\Longrightarrow}} LQ'$ yields the *weak* versions of the corresponding simulations. A located process LP weakly bisimulates LQ with respect to Γ, denoted $LP \mathrel{\dot{\approx}_\Gamma} LQ$ if there exists a weak bisimulation indexed by Γ containing the pair (LP, LQ).

We now prove that $\dot{\sim}$ and $\dot{\approx}$ are reasonable equality relations: they are equivalence relations, they are preserved by injective substitution, and they include structural congruence. The proofs are standard.

Lemma 5.2.1 (Bisimulation is an equivalence relation)
For any closed well-formed type context Γ, $\dot{\sim}_\Gamma$ and $\dot{\approx}_\Gamma$ are equivalence relations.

Lemma 5.2.2 (Injective substitution preserves bisimilarity)
Let Γ be a closed located context and $\rho : \mathsf{dom}(\Gamma) \to \mathcal{X}$ be injective:

1. if $LP_1 \mathrel{\dot{\sim}_\Gamma} LP_2$ then $\rho LP_1 \mathrel{\dot{\sim}_{\rho\Gamma}} \rho LP_2$; and

2. if $LP_1 \mathrel{\dot{\approx}_\Gamma} LP_2$ then $\rho LP_1 \mathrel{\dot{\approx}_{\rho\Gamma}} \rho LP_2$.

Lemma 5.2.3 (Strong bisimulation subsumes structural congruence)
Given a closed located type context Γ, if $\Gamma \vdash LP_1$ and $LP_1 \equiv LP_2$ then $LP_1 \mathrel{\dot{\sim}_\Gamma} LP_2$.

We also prove that the relations $\dot{\sim}$ and $\dot{\approx}$ are themselves strong and weak bisimulation, and that $\dot{\approx}$ subsumes $\dot{\sim}$.

Lemma 5.2.4
The binary relations, $\dot{\sim}$ and $\dot{\approx}$, indexed by closed well-formed located type contexts, are strong and weak bisimulations; moreover, $\dot{\sim} \subseteq \dot{\approx}$.

5.3 Translocating Equivalences

To prove operational relation between processes, it is convenient to use a compositional technique, factoring out irrelevant program context which is common to both and concentrating on their "cores". Many process calculus techniques, such as that given in [San95b], have been formulated for this purpose. The most common way is to work with *operational congruences*: operational relations which are preserved by application of a program context.

In Nomadic π-calculi, the behaviour of location-dependent processes depends on the relative location of agents. Processes which are equivalent under one setting might not be equivalent under another setting; for example, consider LP, LQ defined below. Obviously $LP \dot{\sim}_\Gamma LQ$ whenever $\Gamma \vdash LP, LQ$ and a, b are agents located at the same site s in Γ.

$$\begin{array}{rcl} LP & = & @_a\mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ LQ & = & @_a\mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ P\ \mathbf{else}\ Q' \end{array}$$

The standard equivalence $\dot{\sim}$, defined in Section 5.2, is clearly not resilient to changes in location context caused by process contexts. For example, if the above pair of processes are executed in parallel with a located process $LR = @_b\mathbf{migrate\ to}\ s' \rightarrow \mathbf{0}$, for a different site s', then clearly $LP \mid LR$ does not have the same behaviour as $LQ \mid LR$ under Γ, despite a and b being initially co-located. To obtain a congruence, we need a refined equivalence that takes into account the possibility of such changes of locations by the environment. *Relocators*, ranged over by δ, can be applied to located type contexts in order to relocate agents in such contexts. A valid relocator for (Γ, M), is a type-respecting partial function from M to site names of Γ, formally defined below.

Definition 5.3.1 (Valid relocators)
A relocator δ is said to be *valid for* (Γ, M) if $\mathsf{dom}(\delta) \subseteq M$ and, for all $x \in M$, $\Gamma \vdash x \in \mathtt{Agent}^{\mathtt{m}}$ and $\Gamma \vdash \delta(x) \in \mathtt{Site}$.

We write $\Gamma\delta$ for the result of applying δ to Γ and $\Gamma\delta\beta$ for $(\Gamma\delta)\beta$.

A naive attempt is to check whether a pair of located processes is related by $\dot{\sim}_{\Gamma\delta}$ for all relocators δ (ie. being bisimilar under any relocation of agents in Γ), are congruent.

Unfortunately, this does not suffice. To give a counterexample, consider the following pair of processes:

$$\begin{array}{rcl} LP & = & @_a\ \mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ (\mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2) \\ LQ & = & @_a\ \mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ (\mathbf{iflocal}\ \langle b\rangle c!v\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2') \end{array}$$

These are clearly equivalent for any location context Γ (provided that $\Gamma \vdash LP, LQ$). However, putting such processes in parallel with a located process

$$LR = @_b\textbf{migrate to } s' \to \mathbf{0}$$

with b located $s \neq s'$, shows that they are not congruent, since LR may move the agent b after the first **iflocal** has been tested but before the second, resulting in a situation similar to the previous counterexample.

It is quite clear from the previous example that, in order for a pair of processes to be congruent, they must simulate each other under any relocation of free agents capable of migrating, at any step of their computation. Agents bound in the processes cannot be migrated by the environment, and therefore are not subjected to relocation. Allowing arbitrary relocations would give too strong a notion, though, for some environment may never migrate certain agents. For example, the located processes LP, LQ from the previous example remain equivalent under program contexts $\mathcal{E}[\cdot]$, provided that $\mathcal{E}[\cdot]$ never moves a, b. We need to index standard operational equivalences by a set of agents that the environment may move, which hereinafter shall be referred to as a *translocating* index. We first give some preliminary notions.

Definition 5.3.2 (Translocating Indexed Relation)
A *translocating indexed relation* is a binary relation between $\mathrm{n}\pi_{\mathsf{LD,LI}}$, indexed by closed well-formed located type contexts Γ and sets $M \subseteq \mathsf{mov}(\Gamma)$, where $\mathsf{mov}(\Gamma)$ is the set of names of type $\mathtt{Agent}^{\mathtt{m}}$ in Γ:

$$\begin{array}{lcl}
\mathsf{mov}(\bullet) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{mov}(\Gamma,\ X) & \stackrel{\text{def}}{=} & \mathsf{mov}(\Gamma) \\
\mathsf{mov}(\Gamma,\ x : T) & \stackrel{\text{def}}{=} & \mathsf{mov}(\Gamma) \qquad T \neq \mathtt{Agent}^Z \\
\mathsf{mov}(\Gamma,\ x : \mathtt{Agent}^Z @ s) & \stackrel{\text{def}}{=} & \begin{cases} \mathsf{mov}(\Gamma) \cup \{x\} & Z = \mathtt{m} \\ \mathsf{mov}(\Gamma) & \text{otherwise} \end{cases}
\end{array}$$

The set $\mathsf{agents}(\Gamma)$ is defined similarly to be the set of all names of type $\mathtt{Agent}^Z$ in Γ. Considering relocation of agents at every steps of computation is insufficient for obtaining a congruence, however. To demonstrate this we introduce the following bisimulation; it will not occur later on in the thesis.

Definition 5.3.3 (Strong M-bisimulation)
A translocating indexed relation $\mathcal{S}$ is a *strong M-simulation* if $(LP, LQ) \in \mathcal{S}_{\Gamma}^{M}$ implies:

- $\vdash_{\mathsf{L}} \Gamma$, $\Gamma \vdash LP$ and $\Gamma \vdash LQ$;

- $M \subseteq \mathsf{mov}(\Gamma)$; and
- for any relocator δ valid for (Γ, M), if $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ then there exists LQ' such that $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ and $(LP', LQ') \in \mathcal{S}^{M}_{\Gamma\delta\beta,\Delta}$.

$\mathcal{S}$ is called a *strong M-bisimulation* if all of its indexed relations are symmetric.

This strong M-bisimulation is not a congruence, since it ignores the extrusion of new names to and from the environment by output and input actions. To show why such is the case, consider the following pair of processes.

$$\begin{aligned} LP &= @_b c\mathbf{?}y \rightarrow \ \mathtt{\mathbf{iflocal}}\ \langle y\rangle c\mathbf{!}v\ \mathtt{\mathbf{then}}\ (\mathtt{\mathbf{iflocal}}\ \langle y\rangle c\mathbf{!}v\ \mathtt{\mathbf{then}}\ P_1\ \mathtt{\mathbf{else}}\ P_2) \\ LQ &= @_b c\mathbf{?}y \rightarrow \ \mathtt{\mathbf{iflocal}}\ \langle y\rangle c\mathbf{!}v\ \mathtt{\mathbf{then}}\ (\mathtt{\mathbf{iflocal}}\ \langle y\rangle c\mathbf{!}v\ \mathtt{\mathbf{then}}\ P_1\ \mathtt{\mathbf{else}}\ P_2') \end{aligned}$$

The above processes consist of a free agent b which may receive a name of an agent from the environment via a channel c and, reacting to such a message, the agent b becomes a process similar to the previous counterexample. Let Γ be the located type context given below.

$$s : \mathtt{Site}, \quad s' : \mathtt{Site}, \quad b : \mathtt{Agent}^{\mathtt{s}} @ s, \quad c : \mathtt{\hat{}}^{\mathtt{rw}}\mathtt{Agent}^{\mathtt{s}}$$

Since neither agent b (which is static) nor agent y (received from the environment) are to be relocated, LP, LQ can be related by strong $\mathsf{mov}(\Gamma)$-bisimulation. Yet placing LP, LQ in parallel with

$$LR = \mathtt{\mathbf{new}}\ a : \mathtt{Agent}^{\mathtt{m}} @ s\ \mathtt{\mathbf{in}}\ @_a(\langle b@s\rangle c\mathbf{!}a \mid \mathtt{\mathbf{migrate\ to}}\ s'\ \mathtt{\mathbf{then}}\ P)$$

is sufficient to show that they are not congruent, since one reduction places the pairs $LP|LR$ and $LQ|LR$ in the same situation as the previous counterexample. Note that we can neither simplify the number of **iflocal** tests in LP and LQ, nor let b be mobile, otherwise the $\mathsf{mov}(\Gamma)$-bisimulation relation between LP and LQ will not hold.

Now consider an output of a new-bound agent name a to the environment. Other components in the environment may then send messages to a, but cannot migrate it, so when checking a translocating equivalence we do not need to consider relocation of a. On the other hand, a new agent name received from the environment by an input process is the name of an agent created in the environment, so as in the example above (if created with the mobile capability) it may be migrated at any time. Therefore the translocating index of the bisimulation only needs to be updated when an input action occurs. For this we define the set $M_1 \uplus_\beta M_2$ to be $M_1 \cup M_2$ whenever β is an input, and to be M_1 otherwise.

$$M_1 \uplus_\beta M_2 \quad \stackrel{\mathrm{def}}{=} \quad \begin{cases} M_1 \cup M_2 & \exists a, c, v,\ \beta = @_a c\mathbf{?}v \\ M_1 & \text{otherwise} \end{cases}$$

The notion of translocating bisimulation can therefore be formalised as follows.

Definition 5.3.4 (Translocating Simulations)

1. A translocating indexed relation $\mathcal{S}$ on $\mathrm{n}\pi_{\mathsf{LD,LI}}$ is a *translocating strong simulation* if $(LP, LQ) \in \mathcal{S}^M_\Gamma$ implies the following:

   - $\vdash_{\mathsf{L}} \Gamma$, $\Gamma \vdash LP$ and $\Gamma \vdash LQ$;
   - $M \subseteq \mathsf{mov}(\Gamma)$; and
   - For any relocator δ valid for (Γ, M), if $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ then there exists LQ' such that $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ and $(LP', LQ') \in \mathcal{S}^{M \uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}$.

   $\mathcal{S}$ is called a *translocating strong bisimulation* if all of its indexed relations are symmetric. Two located processes LP and LQ are translocating strongly bisimilar w.r.t. Γ, M, written $LP \dot{\sim}^M_\Gamma LQ$, if there exists a translocating strong bisimulation which when indexed by Γ and M, contains the pair (LP, LQ).

2. Replacing $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ in the final item of this definition with $\Gamma\delta \Vdash LQ \xRightarrow[\Delta]{\hat{\beta}} LQ'$ yields the *weak* version of translocating simulation. A located process LQ weak translocating bisimulates LP w.r.t. Γ, M, denoted $LP \dot{\approx}^M_\Gamma LQ$, if there exists a weak translocating bisimulation which when indexed by Γ, M, contains the pair (LP, LQ).

Some simple examples of translocating bisimulations are the following.

$$@_a\mathtt{iflocal}\ \langle b\rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ Q \quad \dot{\sim}^{M_1}_\Gamma \quad @_a\mathtt{iflocal}\ \langle b\rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ Q'$$
$$@_a\langle b@s\rangle c!v \quad \dot{\approx}^{M_2}_\Gamma \quad @_b c!v$$

where $M_1 \subseteq \mathsf{mov}(\Gamma)/\{a, b\}$ and $M_2 \subseteq \mathsf{mov}(\Gamma)/\{b\}$; we assume that the processes above are well-typed w.r.t. Γ, and that $\Gamma \vdash a@s$ and $\Gamma \vdash b@s$.

In Section 5.3.2, we determine the conditions under which the above relations are preserved by program contexts. Prior to that, we shall show some basic properties of translocating bisimulations which are required for the proof of congruence results, as well as for the correctness proof.

5.3.1 Basic Properties

We now prove that translocating bisimulations are reasonable equality relations: they are equivalence relations, they are preserved by injective substitution, and they include structural congruence. We also prove that the translocating indexed relations $\dot{\sim}$ and $\dot{\approx}$ are themselves strong and weak bisimulation, and that $\dot{\approx}$ subsumes $\dot{\sim}$. The proofs of these lemmas are standard.

Lemma 5.3.1 (Transloc. equivalences are equivalent relations)
For any closed well-formed type context Γ and $M \subseteq \mathsf{agents}(\Gamma)$, $\dot{\sim}_\Gamma^M$ and $\dot{\approx}_\Gamma^M$ are equivalence relations.

Lemma 5.3.2 (Injection preserves translocating equivalences)
Given that Γ is a closed located type context and $\rho : \mathsf{dom}(\Gamma) \to \mathcal{X}$ is injective, we have:

1. if $LP_1 \,\dot{\sim}_\Gamma^M\, LP_2$ then $\rho LP_1 \,\dot{\sim}_{\rho\Gamma}^{\rho M}\, \rho LP_2$; and
2. if $LP_1 \,\dot{\approx}_\Gamma^M\, LP_2$ then $\rho LP_1 \,\dot{\approx}_{\rho\Gamma}^{\rho M}\, \rho LP_2$.

Proof: See Appendix B.6 on page 211. ■

Lemma 5.3.3
The translocating indexed relations $\dot{\sim}$ and $\dot{\approx}$ are strong and weak translocating bisimulations; moreover, $\dot{\sim} \subseteq \dot{\approx}$.

Lemma 5.3.4 (Str. Congruence is a strong translocating bisimulation)
If Γ is a closed located type context with $\Gamma \vdash LP_1$ and $LP_1 \equiv LP_2$ then, for any $M \subseteq \mathsf{mov}(\Gamma)$, $LP_1 \,\dot{\sim}_\Gamma^M\, LP_2$.

An extensible located type context Δ can be added to, or removed from, the index of a translocating relation $LP \dot{\sim}_\Gamma^M LQ$, provided that Δ does not intersect with the names in LP or LQ. This result is used frequently in the proofs of meta-theory as well as the proof of infrastructure correctness.

Lemma 5.3.5 (Strengthening and weakening: transloc. bisimulations)
Let Γ, Θ be closed located type contexts with Θ extensible and $\vdash_{\mathsf{L}} \Gamma, \Theta$. Let $M_1 \subseteq \mathsf{mov}(\Gamma)$ and $M_2 \subseteq \mathsf{mov}(\Theta)$, then if $\Gamma \vdash LP, LQ$ the following hold.

$$\begin{array}{ccc} LP \dot{\sim}_{\Gamma,\Theta}^{M_1 \cup M_2} LQ & \iff & LP \dot{\sim}_{\Gamma}^{M_1} LQ \\ LP \dot{\approx}_{\Gamma,\Theta}^{M_1 \cup M_2} LQ & \iff & LP \dot{\approx}_{\Gamma}^{M_1} LQ \end{array}$$

Proof: See Appendix B.6 on page 211. ■

Note that the above lemma does not hold for strengthening/weakening of site names, for example, consider the following processes.

$$\begin{array}{rcl} LP & = & @_a \mathbf{iflocal}\ \langle b \rangle c\,\mathbf{!}\,[]\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ LQ & = & @_a \mathbf{iflocal}\ \langle b \rangle c\,\mathbf{!}\,[]\ \mathbf{then}\ P\ \mathbf{else}\ Q' \end{array}$$

If these processes are placed under the located type context containing a single site, for example

$$\Gamma \quad = \quad s : \texttt{Site}, \;\; a : \texttt{Agent}^{\texttt{m}}@s, \;\; b : \texttt{Agent}^{\texttt{m}}@s, \;\; c : \texttt{\^{}}^{\texttt{w}}[]$$

then, clearly, $LP \dot{\sim}_{\Gamma}^{\{b\}} LQ$, since no translocation may occur. However, this relation cannot be weakened by a new site name, eg. $LP \dot{\sim}_{\Gamma, s' : \texttt{Site}}^{\{b\}} LQ$ does not hold, since the agent b may be migrated by the environment to s'. Since we only use this lemma for dealing with dynamically created names, for simplicity we restrict our attention to extensible names.

5.3.2 Congruence Properties

We now show that translocating bisimulation is indeed preserved by certain program contexts, involving (from the syntax of LP) parallel composition and **new**-binding. Intuitively, translocating equivalence is preserved by parallel composition if the translocating equivalence of its components allows all the agent movements of other components. More precisely, $LP|LR \;\dot{\sim}_{\Gamma}^{M}\; LQ|LR$ if $LP \;\dot{\sim}_{\Gamma}^{M}\; LQ$ and M contains all agents that may migrate in LR. We generalise this intuition to a pair of equivalent processes, adding **new**-binding, as follows.

Theorem 5.3.6 (Composing translocating bisimulations)
Let Γ, Θ be closed located type contexts, with Θ extensible; moreover, let $M_P, M_Q \subseteq \mathsf{mov}(\Gamma, \Theta)$, and suppose $\mathsf{mayMove}(LQ, LQ') \subseteq M_P$ and $\mathsf{mayMove}(LP, LP') \subseteq M_Q$.

- $LP \;\dot{\sim}_{\Gamma,\Theta}^{M_P}\; LP'$ and $LQ \;\dot{\sim}_{\Gamma,\Theta}^{M_Q}\; LQ'$ implies:

$$\textbf{new}\ \Theta\ \textbf{in}\ (LP \mid LQ) \;\dot{\sim}_{\Gamma}^{M_P \cap M_Q \cap \mathsf{mov}(\Gamma)}\; \textbf{new}\ \Theta\ \textbf{in}\ \big(LP' \mid LQ'\big)\,;\text{and}$$

- $LP \;\dot{\approx}_{\Gamma,\Theta}^{M_P}\; LP'$ and $LQ \;\dot{\approx}_{\Gamma,\Theta}^{M_Q}\; LQ'$ implies:

$$\textbf{new}\ \Theta\ \textbf{in}\ (LP \mid LQ) \;\dot{\approx}_{\Gamma}^{M_P \cap M_Q \cap \mathsf{mov}(\Gamma)}\; \textbf{new}\ \Theta\ \textbf{in}\ \big(LP' \mid LQ'\big)\,.$$

The theorem uses a further auxiliary definition: the set $\mathsf{mayMove}(LP)$ is the set of free agents in LP syntactically containing **`migrate to`**.

$$\begin{array}{lcl}
\mathsf{mayMove}(@_a P) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(P) \\
\mathsf{mayMove}(LP|LQ) & \stackrel{\text{def}}{=} & \mathsf{mayMove}(LP) \cup \mathsf{mayMove}(LQ) \\
\mathsf{mayMove}(\textbf{new}\ \Delta\ \textbf{in}\ LP) & \stackrel{\text{def}}{=} & \mathsf{mayMove}(LP)/\mathsf{dom}(\Delta)
\end{array}$$

$$
\begin{array}{lcl}
\mathsf{mayMove}_a(\mathbf{0}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{mayMove}_a(P|Q) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(P) \cup \mathsf{mayMove}_a(Q) \\
\mathsf{mayMove}_a(\mathtt{new}\ \Delta\ \mathtt{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(P) \\
\mathsf{mayMove}_a(c!v) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{mayMove}_a(c?p \rightarrow P) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(P) \\
\mathsf{mayMove}_a(*c?p \rightarrow P) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(P) \\
\mathsf{mayMove}_a(\mathtt{if}\ v\ \mathtt{then}\ P\ \mathtt{else}\ Q) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(P) \cup \mathsf{mayMove}_a(Q) \\
\mathsf{mayMove}_a(\mathtt{let}\ p = ev\ \mathtt{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(P) \\
\mathsf{mayMove}_a(\mathtt{create}^Z\ b = P\ \mathtt{in}\ Q) & \stackrel{\text{def}}{=} & \mathsf{mayMove}_a(Q) \\
\mathsf{mayMove}_a(\mathtt{migrate\ to}\ s \rightarrow P) & \stackrel{\text{def}}{=} & \{a\} \\
\mathsf{mayMove}_a(\mathtt{iflocal}\ \langle b\rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ Q) & \stackrel{\text{def}}{=} & \mathsf{mayMove}(P) \cup \mathsf{mayMove}(Q)
\end{array}
$$

We may easily prove that if $a \notin \mathsf{mayMove}(LP)$ then LP will not make a transition under $@_a\mathsf{migrate\ to}\ s$ for any s.

Proof Sketch: The proof of this theorem is non-trivial. It deals with derivatives of $\mathtt{new}\ \Theta\ \mathtt{in}\ LP \mid LQ$ w.r.t. Γ, which have the general form of

$$LR_k \quad = \quad \mathtt{new}\ \Theta, \Theta_{comm}\ \mathtt{in}\ (LP_k \mid LQ_k)$$

well-typed w.r.t. $\Gamma, \Theta_{in}, \Theta_{out}$. Here we classify new names bound in the derivative, and those extruded to or from the environment as follows.

- Θ_{comm} consists of names exchanged by communication between LP and LQ. This can be classified further as Θ^{LP}_{comm}, the private names of LP extruded by output actions to LQ, and vice versa for Θ^{LQ}_{comm}.
- Θ_{out} consists of names extruded by output actions to the environment. Again, this can be classified further as Θ^{LP}_{out}, for the names extruded by LP, and vice versa for Θ^{LQ}_{out}.
- Θ_{in} consists of names received from the environment.

Using this classification of names, the set $\mathsf{mov}(\Theta_{in})$ anticipates the movements of agents received from the environment (ie. the context of LR_k), and the set $M_P \cup \mathsf{mov}(\Theta^{LQ}_{comm}, \Theta^{LQ}_{out})$ anticipates the movements of free agents in LQ_k. Since the environment of LP_k comprises LQ_k and the context of LR_k as a whole, the translocating index of the bisimulation relations between LP_k and LP'_k must include the set below.

$$M_{P_k} \quad = \quad M_P \cup \mathsf{mov}(\Theta^{LQ}_{comm}, \Theta^{LQ}_{out}, \Theta_{in})$$

The premises of Theorem 5.3.6 can therefore be generalised in the coinduction as follows.

- $LP_k \dot{\sim}^{M_{P_k}}_{\Gamma,\Theta_{in},\Theta_{out},\Theta_{comm},\Theta} LP'_k$, and $LQ_k \dot{\sim}^{M_{Q_k}}_{\Gamma,\Theta_{in},\Theta_{out},\Theta_{comm},\Theta} LQ'_k$, where M_{Q_k} is defined in the similar way as M_{P_k};
- $\mathsf{mayMove}(LP_k, LP'_k) \subseteq M_Q \cup \mathsf{mov}(\Theta^{LP}_{comm}, \Theta^{LP}_{out})$; and
- $\mathsf{mayMove}(LQ_k, LQ'_k) \subseteq M_P \cup \mathsf{mov}(\Theta^{LQ}_{comm}, \Theta^{LQ}_{out})$.

The proof of this theorem relies on the invariance under labelled transitions of the above premises. The details of this proof can be found in Appendix B.7 on page 215. ■

Note that here we determine a set of agents that may migrate in a process LP by examining its syntax. Alternatively, we might use means of typing for determining this; the author has experimented with the following.

1. If $\Gamma \vdash LP$ then $\mathsf{mayMove}'(LP)$ could be defined as the set $\mathsf{mov}(\Gamma)$. This means that the above theorem would hold only when $M_P = M_Q = \mathsf{mov}(\Gamma, \Theta)$. We reject this solution since, in some cases, we need to work with translocating equivalences which are not resilient to movement of all mobile agents, as illustrated by an example at the end of this section.

2. If $\Gamma \vdash LP$ then $\mathsf{mayMove}''(LP)$ could be defined as the largest subset of $\mathsf{mov}(\Gamma)$ such that $\Delta \vdash LP$, where Δ is obtained from Γ by replacing the capabilities of names in $\mathsf{mayMove}''(LP)$ with $\mathtt{s}$. We reject this solution for the set of potentially mobile agent determined in this way is not preserved under transition. For example, let LP be $@_a(c?x \rightarrow c!x)$, well-typed w.r.t. Γ. In this case, we have $a \notin \mathsf{mayMove}''(LP)$. However, consider the following transition.

$$\Gamma \Vdash LP \xrightarrow{@_a c?a} @_a c!a$$

If $\Gamma \vdash c \in \mathtt{\hat{}^{rw}Agent^{m}}$ then $a \in \mathsf{mayMove}''(@_a c!a)$.

Congruence Congruence relation over located processes can be defined as translocating bisimulations which are resilient to relocation of any mobile agent, and hence will be preserved by any process context.

Definition 5.3.5 (Congruences for Located Processes)

- Located Processes LP and LQ are *strongly congruent* w.r.t. Γ, written $LP \sim_\Gamma LQ$, if $LP \dot{\sim}^{\mathsf{mov}(\Gamma)}_{\Gamma} LQ$.
- Located Processes LP and LQ are *weakly congruent* w.r.t. Γ, written $LP \approx_\Gamma LQ$, if $LP \dot{\approx}^{\mathsf{mov}(\Gamma)}_{\Gamma} LQ$.

To prove the congruence property, we first define *located process contexts*, ranged over by $\mathcal{E}[\cdot]$, as follow:

$$\mathcal{E}[\cdot] \ ::= \ \cdot \ | \ \mathcal{E}[\cdot]|LQ \ | \ \mathbf{new}\ x : \hat{}^{I}T \ \mathbf{in}\ \mathcal{E}[\cdot] \ | \ \mathbf{new}\ x : \mathtt{Agent}^{Z}@s \ \mathbf{in}\ \mathcal{E}[\cdot]$$

We write $\mathcal{E}[LP]$ for the located process obtained by substituting LP for the hole $[\cdot]$ in the process context $\mathcal{E}[\cdot]$.

We also typecheck process contexts, by the typing rules given in Chapter 3 extended with the following rule.

$$\text{(HOLE)}\quad \frac{\vdash \Gamma}{\Gamma \vdash [.]}$$

Typing can determine which placement of processes in a context is sensible. If $\Gamma \vdash \mathcal{E}[\cdot]$ then, for $\mathcal{E}[LP]$ to be well-typed w.r.t. Γ, we need to typecheck LP w.r.t. Γ together with any binders in $\mathcal{E}[.]$ whose scope includes the hole. We define the *extension of* Γ *by* $\mathcal{E}[.]$, written $\mathcal{E}[\Gamma]$, as follows:

$$\begin{array}{rcl@{\qquad}rcl} [\Gamma] & \stackrel{\text{def}}{=} & \Gamma & \mathbf{new}\ x : \hat{}^{I}T \ \mathbf{in}\ \mathcal{E}[\Gamma] & \stackrel{\text{def}}{=} & \mathcal{E}[\Gamma,\ x : \hat{}^{I}T] \\ \mathcal{E}[\Gamma] \mid LQ & \stackrel{\text{def}}{=} & \mathcal{E}[\Gamma] & \mathbf{new}\ x : \mathtt{Agent}^{Z}@s \ \mathbf{in}\ \mathcal{E}[\Gamma] & \stackrel{\text{def}}{=} & \mathcal{E}[\Gamma,\ x : \mathtt{Agent}^{Z}@s] \end{array}$$

Using Theorem 5.3.6, we may easily prove that the operational congruences are indeed preserved by application of any process context.

Theorem 5.3.7 (Congruence: Located Processes)
Given a located proccess context $\mathcal{E}[\cdot]$, such that $\Gamma \vdash \mathcal{E}[\cdot]$, we have:

- If $LP \ \sim_{\mathcal{E}[\Gamma]} \ LQ$ then $\mathcal{E}[LP] \ \sim_{\Gamma} \ \mathcal{E}[LQ]$.
- If $LP \ \approx_{\mathcal{E}[\Gamma]} \ LQ$ then $\mathcal{E}[LP] \ \approx_{\Gamma} \ \mathcal{E}[LQ]$.

Proof: Easily obtained from Theorem 5.3.6 by an induction on the typing derivation of $\Gamma \vdash \mathcal{E}[\cdot]$. ■

Note that, although this operational congruence gives a less restricted congruence result than translocating bisimulations, we still require the latter in many cases. As an example, we may derive the following relation using the techniques related to temporary immobility (see Section 6.4 on page 104).

$$\begin{array}{l} \mathbf{new}\ \mathtt{ack} : \hat{}^{\mathtt{rw}}[],\ \mathtt{currentloc} : \hat{}^{\mathtt{rw}}\mathtt{Site}\ \mathbf{in} \\ \quad @_D\langle a@s\rangle \mathtt{ack!}[] \mid @_a(\llbracket P \rrbracket_a \mid \mathtt{ack?}[] \rightarrow Q \mid \mathtt{Deliverer}) \\ \dot{\approx}^{M}_{\Phi} \quad \mathbf{new}\ \mathtt{ack} : \hat{}^{\mathtt{rw}}[],\ \mathtt{currentloc} : \hat{}^{\mathtt{rw}}\mathtt{Site}\ \mathbf{in} \\ \qquad @_a\mathtt{ack!}[] \mid @_a(\llbracket P \rrbracket_a \mid \mathtt{ack?}[] \rightarrow Q \mid \mathtt{Deliverer}) \end{array}$$

assuming that $\Phi \vdash a@s$ and $a \notin M$. This property will not hold, however, if a can migrate away (ie. $a \in M$), and therefore the above processes are not weakly congruent.

5.4 Other Operational Relations

Strong and weak bisimulations are insufficient for stating and proving the correctness of distributed infrastructures. We require two further operational relations, defined in this section: expansion and coupled simulation.

5.4.1 Expansions

An expansion preorder refines weak bisimulation in an asymmetric manner — informally, to state that a process P expands a process Q is to state that P and Q exhibit the same observable behaviour, but P may have more internal computation. This equivalence was introduced for CCS by Arun-Kumar and Hennessy in [AKH90], where it is used for proving the correctness of two implementations of a FIFO buffer by relating them to a specification, as well as for comparing their efficiency. One of its key technical properties is that expansion relation can be used for "closing" weak bisimulation relations — as in the technique of weak bisimulation "up to" [SM92], adapted to our setting in §6.1.

A definition of expansion uses two refinements of weak simulation: progressing and strict simulation. We adapt their definitions from [Nes96], adding type contexts and translocation.

Definition 5.4.1 (Progressing and Strict Simulation)
A weak translocating simulation $\mathcal{S}$ is called

- *strict* if, for all $(LP, LQ) \in \mathcal{S}_{\Gamma}^{M}$ and valid δ for (Γ, M), $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ implies there exists LQ' such that $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\hat{\beta}} LQ'$ with $(LP', LQ') \in \mathcal{S}_{\Gamma\delta\beta,\Delta}^{M \uplus_{\beta} \mathsf{mov}(\Delta)}$.
- *progressing* if, for all $(LP, LQ) \in \mathcal{S}_{\Gamma}^{M}$ and valid δ for (Γ, M), $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ implies there exists LQ' such that $\Gamma\delta \Vdash LQ \xRightarrow[\Delta]{\beta} LQ'$ with $(LP', LQ') \in \mathcal{S}_{\Gamma\delta\beta,\Delta}^{M \uplus_{\beta} \mathsf{mov}(\Delta)}$; and

LQ is said to *progressing simulate* (or *strictly simulate*) LP w.r.t. Γ, M if there exists a progressing simulation $\mathcal{S}$ (or a strict simulation) such that $(LP, LQ) \in \mathcal{S}_{\Gamma}^{M}$.

$$\begin{tikzcd}
\Gamma \Vdash LP \arrow[rr, "\beta", "\Delta"'] \arrow[d, dash, "\mathcal{S}_{\Gamma}^{M}"] & & \Gamma\beta, \Delta \Vdash LP_1 \arrow[d, dotted, dash, "\mathcal{S}_{\Gamma\beta,\Delta}^{M \uplus_\beta \mathsf{mov}(\Delta)}"] \\
\Gamma \Vdash LQ \arrow[rr, dotted, "\hat{\beta}", "\Delta"'] & & \Gamma\beta, \Delta \Vdash LQ_1
\end{tikzcd}$$

(a) strict simulation

$$\begin{tikzcd}
\Gamma \Vdash LP \arrow[rr, "\beta", "\Delta"'] \arrow[d, dash, "\mathcal{S}_{\Gamma}^{M}"] & & \Gamma\beta, \Delta \Vdash LP_1 \arrow[d, dotted, dash, "\mathcal{S}_{\Gamma\beta,\Delta}^{M \uplus_\beta \mathsf{mov}(\Delta)}"] \\
\Gamma \Vdash LQ \arrow[rr, Rightarrow, dashed, "\beta", "\Delta"'] & & \Gamma\beta, \Delta \Vdash LQ_1
\end{tikzcd}$$

(b) progressing simulation

The above diagrams show progressing and strict simulations. Informally, LQ strictly simulates LP means that LQ weakly simulates LP, but LQ never introduce more internal steps and may ignore the silent transitions of LP. On the other hand, LQ progressing simulates LP means that LQ weakly simulates LP, but LQ introduces more internal steps and never ignores a silent action, hence the absence of $\hat{}$ in the weak transition of LQ in the definition. The definition of expansion simply makes use of these two refinements.

Definition 5.4.2 (Expansion *à la* [Nes96])
An indexed binary relation $\mathcal{S}$ is a *translocating expansion* if $\mathcal{S}$ is a strict simulation and $\mathcal{S}^{-1}$ is a progressing simulation.

LP translocating expands LQ w.r.t. Γ under M, written $LP \dot{\succeq}_{\Gamma}^{M} LQ$, if there exists an expansion $\mathcal{S}$ with $(LP, LQ) \in \mathcal{S}_{\Gamma}^{M}$. Moreover, if $LP \dot{\succeq}_{\Gamma}^{\mathsf{mov}(\Gamma)} LQ$ then LP and LQ are said to be related by *expansion congruence*, written $LP \succeq_{\Gamma} LQ$.

We prove some basic properties of translocating expansion: that it is preserved by injective substitution, and the strengthening and weakening lemma. The proofs of these lemmas are similar to those for translocating bisimulations.

Lemma 5.4.1 (Injection preserves translocating equivalences)
Given that Γ is a closed located type context and $\rho : \mathsf{dom}(\Gamma) \to \mathcal{X}$ is injective, if $LP_1 \dot{\succeq}_{\Gamma}^{M} LP_2$ then $\rho LP_1 \dot{\succeq}_{\rho\Gamma}^{\rho M} \rho LP_2$.

Lemma 5.4.2 (Strengthening and weakening: transloc. expansions)
Let Γ, Θ be closed located type contexts with Θ is extensible and $\vdash_{\mathsf{L}} \Gamma, \Theta$. Let $M_1 \subseteq \mathsf{mov}(\Gamma)$ and $M_2 \subseteq \mathsf{mov}(\Theta)$, then if $\Gamma \vdash LP, LQ$ the following hold.

$$LP \dot{\succeq}_{\Gamma,\Theta}^{M_1 \cup M_2} LQ \quad \iff \quad LP \dot{\succeq}_{\Gamma}^{M_1} LQ$$

Adding translocation allows compositional reasoning using expansions. We depend on a congruence result, analogous to Theorem 5.3.6, for expansion. Its proof is similar to that of Theorem 5.3.6.

Theorem 5.4.3 (Composing translocating expansion)
Let Γ, Θ be closed located type contexts, with Θ extensible; moreover, let $M_P, M_Q \subseteq \mathsf{mov}(\Gamma, \Theta)$, $\mathsf{mayMove}(LQ, LQ') \subseteq M_P$, $\mathsf{mayMove}(LP, LP') \subseteq M_Q$, and suppose $LP \,\dot{\succeq}^{M_P}_{\Gamma,\Theta}\, LP'$ and $LQ \,\dot{\succeq}^{M_Q}_{\Gamma,\Theta}\, LQ'$, then we have:

$$\mathbf{new}\ \Theta\ \mathbf{in}\ (LP \mid LQ) \ \dot{\succeq}^{M_P \cap M_Q \cap \mathsf{mov}(\Gamma)}_{\Gamma}\ \mathbf{new}\ \Theta\ \mathbf{in}\ \big(LP' \mid LQ'\big)$$

As in Theorem 5.3.7, we may easily prove that the expansion congruence is indeed preserved by application of any located process context.

Theorem 5.4.4 (Expansion congruence: Located Processes)
For any context $\mathcal{E}[\cdot]$, if $\Gamma \vdash \mathcal{E}[\cdot]$ and $LP \ \succeq_{\mathcal{E}[\Gamma]}\ LQ$ then $\mathcal{E}[LP] \ \succeq_{\Gamma}\ \mathcal{E}[LQ]$.

Proof: Easily obtained from Theorem 5.4.3 by an induction on the typing derivation of $\Gamma \vdash \mathcal{E}[\cdot]$. ■

5.4.2 Coupled Simulations

The bisimulation requirement is sometimes too strong for intuitively correct protocols. In particular, some protocols involve making internal choices, resulting in partially committed states — which cannot be related by a bisimulation relation, either to their initial or to their committed states.

As an example, consider the encoding $\mathcal{C}[\![LP]\!]$ of an agent a which sends message $c!v$ to agent b at the current site of a, and in parallel visits the sites s_1 and s_2 (in any order).

$$LP \ \stackrel{\text{def}}{=}\ @_a(\langle b\rangle c!v \mid (\mathbf{migrate\ to}\ s_1 \to \mathbf{0}) \mid (\mathbf{migrate\ to}\ s_2 \to \mathbf{0}))$$

Assuming a and b are initially at the same site, parts of the derivation trees of LP and $\mathcal{C}[\![LP]\!]$ can be represented as in Figure 5.1. If the **migrate to** s_1 process in $\mathcal{C}[\![LP]\!]$ successfully acquires the local lock (a partial commitment step) the resulting process (LQ_{1p} in Figure 5.1) does not correspond exactly to any state of LP. LQ_{1p} cannot correspond to LP_1 since, executing $\langle b\rangle c!v$ at this point means that $c!v$ will reach b (which is not the case for node LP_1); it cannot correspond to LP either, since we know that a will eventually end up in s_2.

Testing equivalence [Hen88] is a weaker equivalence which does not suffer from this problem. This equivalence presupposes some precisely defined class of *tests*, and two transition systems are equivalent if they can pass exactly the same test; different classes of tests yield different equivalences. By including traces and deadlock detection among the tests, an interesting

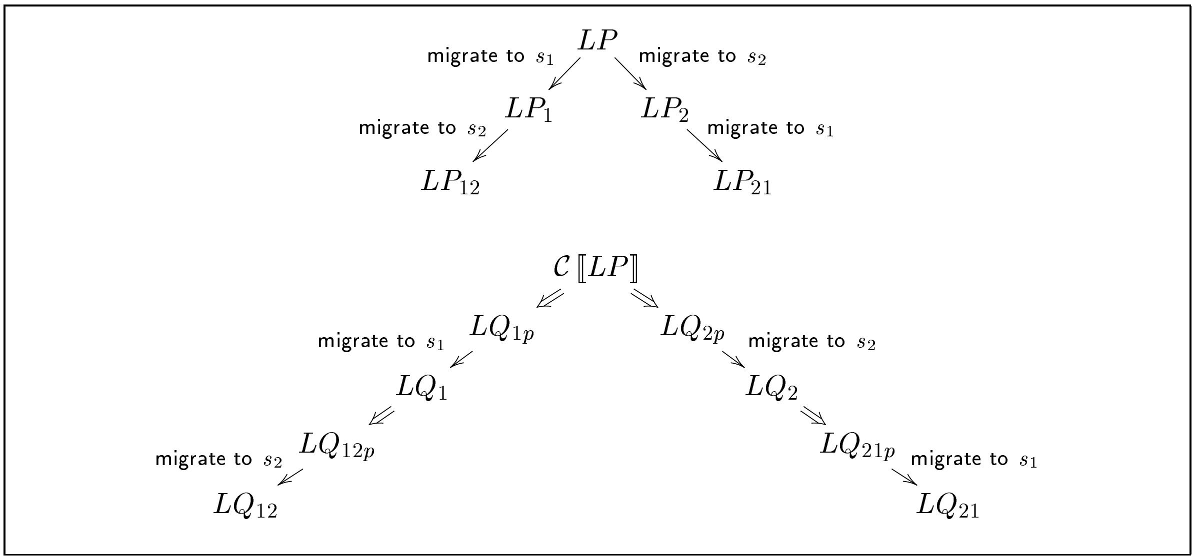


Figure 5.1: An example of partial committed state

equivalence that coincides with the "failure" equivalence of CSP [Hoa85] can be obtained. This equivalence is weaker than bisimilarity; so it can equate transition systems which differ in the way internal choices are resolved. However, to establish that testing equivalence holds it is necessary to perform a case analysis over all possible sequence of transitions. This is a much more difficult task than analysing single transitions, and can be awkward when analysing processes with a large number of transitions.

The coupled simulation of Parrow and Sjodin [PS92] allows the behaviour of processes containing different internal choices to be related, yet retains the co-inductive proof technique enjoyed by bisimulation equivalence. It relaxes the bisimulation clauses by allowing related processes to *mutually simulate* each other via two contrary simulation relations, yet requires them to be *coupled* in some way. Several candidates have been presented for what it means to be coupled. No coupling at all will lead to an equivalence which is finer than trace equivalence [Hoa85], but is coarser than bisimulation. A non-trivial notion of coupling was based on the property of *stability* [PS92], requiring the coincidence of the two simulations whenever processes are stable, ie. when they cannot commit a τ action. This style induces a relation which is an equivalence only for convergent processes; it has been proven to be strictly weaker than bisimulation and strictly stronger than testing equivalence [PS92]. Following [Nes96], we use a generalisation for divergent processes, as suggested in [vG93] and [PS94a], where coupling requires the ability of a simulating process to evolve into a simulated process by internal actions.

The definition of coupled simulation, adapted from Nestmann's notion by adding located type contexts, is given below.

Definition 5.4.3 (Coupled Simulation *à la* [Nes96])
A pair of binary relations on $\mathrm{n}\pi_{\mathsf{LD,LI}}$, indexed by closed located type contexts, $(\mathcal{S}_1, \mathcal{S}_2)$ is a *coupled simulation* if:

- $\mathcal{S}_1$ and $(\mathcal{S}_2)^{-1}$ are weak simulations;
- if $(LP, LQ) \in (\mathcal{S}_1)_\Gamma$ then there exists LQ' such that $\Gamma \Vdash LQ \Longrightarrow LQ'$ and $(LP, LQ') \in (\mathcal{S}_2)_\Gamma$; and
- if $(LP, LQ) \in (\mathcal{S}_2)_\Gamma$ then there exists LP' such that $\Gamma \Vdash LP \Longrightarrow LP'$ and $(LP', LQ) \in (\mathcal{S}_1)_\Gamma$.

Two processes LP, LQ are *coupled similar* w.r.t. Γ, written $LP \leftrightharpoons_\Gamma LQ$, if they are related by both components of some coupled simulation.

Intuitively "LQ coupled simulates LP" means that "LQ is at most as committed as LP" with respect to internal choices and that LQ may reduce to a state LQ' which is at least as committed as LP, ie. where LP coupled simulates LQ'.

Note that in this thesis coupled simulation will be used for relating whole systems, which cannot be placed in any program context. For this reason, we do not need to incorporate translocation into the definition above.

Considering the previous example, we should be able to relate the partial-committed process LQ_{1p} to LP and LP_1 by a coupled simulation relation. Unfortunately, however, this is not possible without further restriction. Consider the following relations $\mathcal{S}_1, \mathcal{S}_2$.

$$\mathcal{S}_1 = \left\{ \begin{array}{r} (LP, \mathcal{C}[\![LP]\!]),\ (LP_1, LQ_1),\ (LP_{12}, LQ_{12p}),\ (LP_{12}, LQ_{12}), \\ (LP_2, LQ_2),\ (LP_{21}, LQ_{21p}),\ (LP_{21}, LQ_{21}) \end{array} \right\}$$

$$\mathcal{S}_2 = \left\{ \begin{array}{r} (LP, \mathcal{C}[\![LP]\!]),\ (LP_1, LQ_{1p}),\ (LP_1, LQ_1),\ (LP_{12}, LQ_{12p}),\ (LP_{12}, LQ_{12}), \\ (LP_2, LQ_{2p}),\ (LP_2, LQ_2),\ (LP_{21}, LQ_{21p}),\ (LP_{21}, LQ_{21}) \end{array} \right\}$$

It is not difficult to show that $\mathcal{S}_1$ and $(\mathcal{S}_2)^{-1}$ are weak simulation. Now for $(\mathcal{S}_1, \mathcal{S}_2)$ to be a coupled simulation, the coupling condition requires that there exists LQ' such that

$$\Phi \Vdash LQ_{1p} \Longrightarrow LQ'$$

and the pair (LP, LQ') is related by $\mathcal{S}_1$. The above reduction cannot hold, since in order that LQ_1 can reach its fully-committed state LQ_1 it needs to perform an observable action $\mathsf{migrate\ to}\ s$.

It might be thought that migration should be made a silent action, so that the above problem would not occur and the coupling condition can be met. This is untrue. Considering the simulation relation between LP_1 and LQ_{1p}. We see that the behaviour of such processes are compared w.r.t. *different* located type contexts: Γ for LQ_{1p} and $(\Gamma \oplus a \mapsto s)$ for LP_1, since a has migrated. This is not possible for our operational equivalences, and to avoid dealing with such a situation, we only work with systems in which all observable agents are static. Such a restriction does not reduce the expressiveness of programs, however. One may imagine each of these observable agents to be assigned to each site, acting as an I/O manager which forwards keyboard inputs to agents, and sends messages to the standard output when requested by an agent. Mobile agents, which can be dynamically created by a program (and hence are **new**-bound), may then interact with the users via these I/O managers.

We prove two properties of coupled simulation: that it is an equivalence relation, and that it subsumes translocating expansion. These properties are required in the proof of correctness for relating the operational correspondence result between the source and the intermediate languages to that between the target and the intermediate languages.

Lemma 5.4.5 (Coupled simulation is an equivalence relation)
For any closed well-formed type context Γ, $\risingdotseq_\Gamma$ is an equivalence relation.

Proof: Standard. Reflexivity and symmetry are easy to establish. Transitivity, however, is non-trivial.

Supposing $LP \risingdotseq_\Gamma LQ$ and $LQ \risingdotseq_\Gamma LR$, then there exist two pairs of coupled simulation $(\mathcal{S}_1, \mathcal{T}_1)$ and $(\mathcal{S}_2, \mathcal{T}_2)$ such that

- $(LP, LQ) \in (\mathcal{S}_1)_\Gamma \cap (\mathcal{T}_1)_\Gamma$; and
- $(LQ, LR) \in (\mathcal{S}_2)_\Gamma \cap (\mathcal{T}_2)_\Gamma$.

We construct two relations $\mathcal{S}, \mathcal{T}$ as follows.

$$\begin{array}{rcl} \mathcal{S}_\Gamma & = & \{(LP, LR) \mid \exists LQ \,.\, (LP, LQ) \in (\mathcal{S}_1)_\Gamma \wedge (LQ, LR) \in (\mathcal{S}_2)_\Gamma\} \\ \mathcal{T}_\Gamma & = & \{(LP, LR) \mid \exists LQ \,.\, (LP, LQ) \in (\mathcal{T}_1)_\Gamma \wedge (LQ, LR) \in (\mathcal{T}_2)_\Gamma\} \end{array}$$

It is easy to show that $\mathcal{S}$ and $\mathcal{T}^{-1}$ are weak simulations. To prove that $(\mathcal{S}, \mathcal{T})$ is a coupled simulation, we need to show that the above relations satisfy the coupled condition.

Let $(LP', LR') \in \mathcal{S}_{\Gamma'}$, ie. there exists LQ' such that $(LP', LQ') \in (\mathcal{S}_1)_{\Gamma'}$ and $(LQ', LR') \in$

$(\mathcal{S}_2)_{\Gamma'}$. Graphically, we need to show that:

$$\begin{tikzcd}
\Gamma \Vdash LP' \arrow[d, dash, "\mathcal{S}'_{\Gamma'}"'] \arrow[drr, dotted, dash, "\mathcal{T}'_{\Gamma'}"] & & \\
\Gamma \Vdash LR' \arrow[rr, Rightarrow, dashed] & & LR'''
\end{tikzcd}$$

By considering the pairs of coupled simulation $(\mathcal{S}_1, \mathcal{T}_1)$ and $(\mathcal{S}_2, \mathcal{T}_2)$, we have:

$$\begin{tikzcd}
\Gamma \Vdash LP' \arrow[d, dash, "(\mathcal{S}'_1)_{\Gamma'}"'] \arrow[dr, dotted, dash, "(\mathcal{T}'_1)_{\Gamma'}"] & & \\
\Gamma \Vdash LQ' \arrow[r, Rightarrow, dashed] \arrow[d, dash, "(\mathcal{S}'_2)_{\Gamma'}"'] & LQ'' \arrow[d, dotted, dash, "(\mathcal{S}'_2)_{\Gamma'}"'] \arrow[dr, dotted, dash, "(\mathcal{T}'_2)_{\Gamma'}"] & \\
LR' \arrow[r, Rightarrow, dashed] & LR'' \arrow[r, Rightarrow, dashed] & LR'''
\end{tikzcd}$$

More precisely:

- Since $(\mathcal{S}_1, \mathcal{T}_1)$ is a coupled simulation, there exists LQ'' such that $\Gamma' \Vdash LQ' \Longrightarrow LQ''$ and $(LP', LQ'') \in (\mathcal{T}_1)_{\Gamma'}$.
- Since $\mathcal{S}_2$ is a weak bisimulation, there exists LR'' such that $\Gamma' \Vdash LR' \Longrightarrow LR''$ and $(LQ'', LR'') \in (\mathcal{S}_2)_{\Gamma'}$.
- Since $(\mathcal{S}_2, \mathcal{T}_2)$ is a coupled simulation, there exists LR''' such that $\Gamma' \Vdash LR'' \Longrightarrow LR'''$ and $(LQ'', LR''') \in (\mathcal{T}_2)_{\Gamma'}$.

This means there exists LR''' such that $\Gamma' \Vdash LR'' \Longrightarrow LR'''$ and $(LP', LR''') \in \mathcal{T}_{\Gamma'}$. Similarly the other coupling condition can be derived. Hence the lemma. ■

Lemma 5.4.6 (Coupled simulation subsumes expansion)
$LP \dot{\succeq}^M_\Gamma LQ$ implies $LP \fallingdotseq_\Gamma LQ$.

Proof: $LP \dot{\succeq}^M_\Gamma LQ$ implies there exists a translocating expansion $\mathcal{S}$ such that $(LP, LQ) \in \mathcal{S}^M_\Gamma$. It is easy to prove that the pair $(\mathcal{S}, \mathcal{S})$ is a coupled simulation. We omit the details. ■

Chapter 6

Proof Techniques

This chapter investigates proof techniques which are required for proving the example infrastructure correct. Many of these techniques are adapted from the π-calculus, adding translocation. Section 6.1 discusses the technique of bisimulation "up to". In particular, we give a definition of an *expansion up to expansion.* This allows reduction of the size of the relation used for establishing an expansion by omitting processes that are related (in some "stronger" way) to other processes in such a relation. Section 6.2 defines a number of channel usage disciplines which allow, among other things, expansions to be derived from computational steps that are essentially functional. Section 6.3 digresses to a discussion of determinacy and confluence in process calculi. We argue that proving that a process is confluent is non-trivial in general, and especially in Nomadic π-calculi where we need to take into account type contexts and translocation. As an alternative, we give a definition of deterministic processes, whose reductions give rise to translocating expansions. Next we introduce a novel technique, identifying properties of agents that are *temporary immobile*, waiting on a lock somewhere in the system. In Section 6.4, we make this precise, and show that a deterministic reduction, when placed in parallel with a temporarily immobile process, gives rise to an expansion. Finally, Section 6.5 gives an encoding of the finite maps used in the example infrastructure and proves its correctness.

6.1 Techniques of Bisimulation "Up To"

The most straightforward way of proving an expansion (or similar coinductive relation) between a pair of Nomadic π processes is to construct a translocating expansion relation containing the pair. The task of constructing a translocating indexed relation and verifying that it meets all the requirements is only manageable for processes whose structures are simple, or whose possible set of transitions is relatively small. In practice, the relation can be quite complex and the case analysis involved becomes quite challenging.

The technique of bisimulation "up to" [SM92] offers a way of reducing the size of bisimulation relations: One may define a relation which is a bisimulation only when closed up under some stronger relation, so reducing the required proof work. We adapt expansion up to expansion from [SM92], adding translocation. The definition is given below.

Definition 6.1.1 (Expansion up to Expansion)
A translocating indexed relation $\mathcal{S}$ on $\mathrm{n}\pi_{\mathsf{LD,LI}}$ is a *translocating expansion up to expansion* if, for all closed located type context Γ and $M \subseteq \mathsf{mov}(\Gamma)$, $LP\mathcal{S}_\Gamma^M LQ$ implies the following.

- For any valid δ for (Γ, M), $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ implies there exists LQ' such that
  - $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\hat{\beta}} LQ'$ and;
  - $LP' \;\dot{\succeq}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}\; \mathcal{S}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}\; \dot{\sim}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}\; LQ'$.
- For any valid δ for (Γ, M), $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ implies there exists LP' such that
  - $\Gamma\delta \Vdash LP \overset{\beta}{\underset{\Delta}{\Longrightarrow}} LP'$ and;
  - $LP' \;\dot{\succeq}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}\; \mathcal{S}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}\; \dot{\sim}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}\; LQ'$.

The relations between processes which are related by an expansion up to expansion must satisfy two diagrams below.

$$\begin{tikzcd}
\Gamma \Vdash LP \arrow[r, "\beta", "\Delta"'] \arrow[d, dash, "\mathcal{S}_\Gamma^M"] & \Gamma, \Delta \Vdash LP' \arrow[d, dotted, dash, "\dot{\succeq}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\beta,\Delta}\; \mathcal{S}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\beta,\Delta}\; \dot{\sim}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\beta,\Delta}"] \\
\Gamma \Vdash LQ \arrow[r, dotted, "\hat{\beta}", "\Delta"'] & \Gamma, \Delta \Vdash LQ'
\end{tikzcd}$$

$$\begin{tikzcd}
\Gamma \Vdash LP \arrow[r, Rightarrow, dashed, "\beta", "\Delta"'] \arrow[d, dash, "\mathcal{S}_\Gamma^M"] & \dot{\succeq}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\beta,\Delta}\; \Gamma, \Delta \Vdash LP' \arrow[d, dotted, dash, "\mathcal{S}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\beta,\Delta}\; \dot{\sim}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\beta,\Delta}"] \\
\Gamma \Vdash LQ \arrow[r, "\beta", "\Delta"'] & \Gamma, \Delta \Vdash LQ'
\end{tikzcd}$$

Processes which are related by an expansion up to an expansion are indeed related by an expansion.

Lemma 6.1.1 (Expansion up to expansion)
If $\mathcal{S}$ is a translocating expansion up to expansion and $LP\ \mathcal{S}_{\Gamma}^{M}\ LQ$ then $LP\ \dot{\succeq}_{\Gamma}^{M}\ LQ$.

Proof: Standard. Supposing $\mathcal{S}$ is a translocating expansion up to expansion, we construct an (indexed) relation $\mathcal{R}$ such that $\mathcal{R}_{\Phi}^{M'}$ is defined as follows:

$$\mathcal{S}_{\Phi}^{M'} \quad \stackrel{\text{def}}{=} \quad \{(LP', LQ') \mid \exists LP'', LQ'' \;.\; LP' \dot{\succeq}_{\Phi}^{M'} LP'' \;\wedge\; LP'' \mathcal{S}_{\Phi}^{M'} LQ'' \;\wedge\; LQ' \dot{\sim}_{\Phi}^{M'} LQ''\}$$

We may then check that $\mathcal{S}$ is a translocating expansion. Hence the lemma (details omitted). ■

Fournet and Gonthier advocate proving bisimulation using *decreasing diagrams* [Fou98]: a technique which reduces bisimulation diagrams to a series of decreasing tiles that can be proved separately in smaller, easier to reuse, lemmas. It is shown that many of the standard up to techniques are easily and uniformly subsumed by the decreasing diagram method. This technique could be generalised for Nomadic π-calculi. However, the fact that we are dealing with the labelled transition semantics, translocating and typed operational relations may introduce some difficulties, since the base meta-theory (from [vO94]) does not deal with indexed relations between processes. Expansion up to expansion is sufficient for the proof of infrastructure correctness; we defer a full investigation of applying the decreasing diagram technique to Nomadic π-calculi to future work.

6.2 Channel-Usage Disciplines

In the π-calculus, channels are first-class data: they can be used for sending, or receiving messages, but also as parts of messages themselves. This flexibility, although giving rise to a great expressive power, can sometimes make programs difficult to read and debug. Programmers often impose certain disciplines on usage of channels to ensure that their programs are ultimately free from race conditions, unintended deadlocks, and other kinds of protocol violations. In the process calculi literature, there are many static analysis techniques capable of detecting such errors, eg. I/O subtyping [PS96], uniform receptiveness [San99], linear types [KPT96], and a generic type system of Igarashi and Kobayashi [IK01]. These techniques mostly involve enriching channel types; typechecking can then be employed for capturing useful properties of programs. This section shows how these techniques can be adapted to Nomadic π-calculi, although to avoid meta-theoretic complexity, they are now based on the syntactic structure of process rather than embellishing the current type system.

We begin in Section 6.2.1 by studying *non-sendable* channels, which cannot be transmitted by communication and therefore enable all possible usages of such channels to be directly determined from the syntax. Sections 6.2.2 and 6.2.3 are devoted to channels which behave as *functions* in the following senses:

- a communication step which consumes an output on such channels guarantees a deterministic result; and
- each such channel has an input which is always present.

We demonstrate two ways which this can be accomplished, and prove that computational steps that are essentially functional give rise to expansions. In Section 6.2.4, we conclude by studying channels that are only used locally within agents, allowing scope readjustment.

6.2.1 Non-sendability

A name is said to be *non-sendable* if it is never transferred from one process to another. For example, x is non-sendable in $@_a(x\,\mathtt{!}[] \mid P)$ (provided eg. $x \notin \mathsf{fv}(P)$), but not in $@_a c\,\mathtt{!}\,x$, since x can be sent to the environment. A formal definition of non-sendability can therefore be given below.

Definition 6.2.1 (Non-Sendability)
A channel c is said to be *non-sendable* in LP if $c \notin \mathsf{chObj}(LP)$, where the definition of $\mathsf{chObj}(LP)$ is given in Figure 6.1.

Note that for a channel c to be non-sendable, it must not occur in the expression ev of the process $\mathbf{let}\ p = ev\ \mathbf{in}\ P$. This prevents c from being involved in the substitution applied to P in the $\mathbf{let}$ process, as in $@_a\mathbf{let}\ x = c\ \mathbf{in}\ c'\,\mathtt{!}\,x$.

In [YH99b], Yoshida and Hennessy study locality of channels in a distributed process calculi $\mathrm{D}\pi\lambda$. A type system is formulated to guarantee that, at any one time, the input capability of channels resides at exactly one location. Sendable/non-sendable subtyping is incorporated in such a type system to ensure that locality is preserved by communication. Likewise, we use non-sendability to ensure that subsequent channel disciplines are not violated by channel communication. The following lemma shows that non-sendability is preserved by labelled transitions. The case where the non-sendable channel is received from the environment via an input action must be excluded, however. Such pathological cases could be easily handled by the labelled transition semantics of Nomadic π-calculi if sendable/non-sendable subtyping were to be incorporated.

$$
\begin{array}{lcl}
\mathsf{chObj}(@_a P) & \stackrel{\text{def}}{=} & \mathsf{chObj}(P) \\
\mathsf{chObj}(LP|LQ) & \stackrel{\text{def}}{=} & \mathsf{chObj}(LP) \cup \mathsf{chObj}(LQ) \\
\mathsf{chObj}(\mathbf{new}\ \Delta\ \mathbf{in}\ LP) & \stackrel{\text{def}}{=} & \mathsf{chObj}(LP)/\mathsf{dom}(\Delta) \\[1ex]
\mathsf{chObj}(\mathbf{0}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{chObj}(P|Q) & \stackrel{\text{def}}{=} & \mathsf{chObj}(P) \cup \mathsf{chObj}(Q) \\
\mathsf{chObj}(\mathbf{new}\ \Delta\ \mathbf{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{chObj}(P)/\mathsf{dom}(\Delta) \\
\mathsf{chObj}(c\,\mathbf{!}\,v, \langle b@?\rangle c\,\mathbf{!}\,v) & \stackrel{\text{def}}{=} & \mathsf{fv}(v) \\
\mathsf{chObj}(c\mathbf{?}p \rightarrow P, \mathbf{*}c\mathbf{?}p \rightarrow P) & \stackrel{\text{def}}{=} & \mathsf{chObj}(P)/\mathsf{fv}(p) \\
\mathsf{chObj}(\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q) & \stackrel{\text{def}}{=} & \mathsf{chObj}(P) \cup \mathsf{chObj}(Q) \\
\mathsf{chObj}(\mathbf{let}\ p = ev\ \mathbf{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{fv}(ev) \cup \mathsf{chObj}(P)/\mathsf{fv}(p) \\
\mathsf{chObj}(\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q) & \stackrel{\text{def}}{=} & (\mathsf{chObj}(P) \cup \mathsf{chObj}(Q))/\{b\} \\
\mathsf{chObj}(\mathbf{migrate\ to}\ s \rightarrow P) & \stackrel{\text{def}}{=} & \mathsf{chObj}(P) \\
\mathsf{chObj}(\mathbf{iflocal}\ \langle b\rangle c\,\mathbf{!}\,v\ \mathbf{then}\ P\ \mathbf{else}\ Q) & \stackrel{\text{def}}{=} & \mathsf{fv}(v) \cup \mathsf{chObj}(P) \cup \mathsf{chObj}(Q)
\end{array}
$$

Figure 6.1: Sendable names

Lemma 6.2.1 (Non-sendability is preserved by LTS)
Given that c is non-sendable in LP, if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ and $\beta \neq @_a x\mathbf{?}v$ with $c \in \mathsf{fv}(v)$ then c is non-sendable in LQ.

Proof: An induction on the transition derivations of process in which c is non-sendable. ■

6.2.2 Access Restrictions

An early form of the π-calculus [MPW92] allowed parametric process definitions of the general form $K(\vec{x}) \stackrel{\text{def}}{=} P_K$. Executing such definitions invokes a structural congruence law $K(\vec{y}) \equiv P_K\{\vec{y}/\vec{x}\}$, which in effect makes a function call, ie. instantiates the definition of P_K with appropriate parameters. Milner [Mil93b] shows that such process definitions can be easily encoded into π-calculus with replication (provided that the number of such definitions are finite). For example, a single recursive definition with a single parameter $A(x) \stackrel{\text{def}}{=} P$ in a process S can be translated to:

$$(\nu c)(\hat{S} \mid !c(x).\hat{P}) \qquad (*)$$

where $\hat{Q}$ is obtained from Q by replacing each call $A(z)$ just by $\bar{c}z$ (assuming c is not free in S, P). The *replicated resource* $!c(x).\hat{P}$ reacts to the output $\bar{c}z$ by instantiating a new copy of P. Structure similar to that in $(*)$ is widely employed in programs expressed in the π-calculus, encodings of data structures etc. The programming language Pict [PT00] includes this style of parametric process definition as a derived form.

A communication via c of the process in $(*)$ can be considered functional, since its result is deterministic, guaranteed by the fact that there exists a unique process which may react to outputs on c. To formalise this, we need a way of checking that c is not used for input in $\hat{S}$ and $\hat{P}$. A channel c is said to be *output-only* in a process LP if LP only uses c for output. As in [PS96], we use input/output subtyping for ensuring restricted usage of a channel in a process.

Definition 6.2.2 (Output-only)
Let LP be a located process and Γ be a closed type context such that $\Gamma \vdash LP$. A channel c, with $\Gamma(c) = \hat{\ }^{I}T$, is output-only in LP w.r.t. Γ if $\Delta \vdash LP$, where Δ is obtained from Γ by replacing the capability of c by $\mathtt{w}$.

Replacing LP with P and $\Gamma \vdash LP$ with $\Gamma \vdash_a P$, for some a, we obtain the definition of c being output-only in basic processes.

The restriction of usage of a channel in a process is an invariant under labelled transitions; this property is formally stated below. As for Lemma 6.2.1, we exclude the cases where the channel which is output-only is received from the environment via an input action.

Lemma 6.2.2 (LTS preserves access discipline)
If c is output-only in LP w.r.t. Γ with $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$, $\beta \neq @_a x\mathtt{?}v$ and $c \in \mathsf{fv}(v)$ then c is output-only in LQ w.r.t. Γ, Δ.

Proof: An induction on transition derivations. ■

The structure in $(*)$ can be adapted to Nomadic π-calculi, adding location annotations. Since in Nomadic π only co-located inputs and outputs on the same channel may synchronise, the function definition and function calls must be located at the same agent (any agent may remotely invoke the function c at another agent a by sending an inter-agent message $c\mathtt{!}v$ to a). We prove that communicating on a functional channel gives rise to an expansion. The formal statement is given below.

Lemma 6.2.3 (Functional computation yields an expansion)
Given a closed located type context Γ with

- $(\Gamma,\ c : \hat{\ }^{\mathtt{rw}}T) \vdash LP \mid @_a c\mathtt{!}v$, $(\Gamma,\ c : \hat{\ }^{\mathtt{rw}}T) \vdash p \in T \triangleright \Delta$, and $(\Gamma,\ c : \hat{\ }^{\mathtt{rw}}T, \Delta) \vdash_a P$;
- c is output-only in P w.r.t. $\Gamma,\ c : \hat{\ }^{\mathtt{rw}}T, \Delta$ and in LP w.r.t. $\Gamma,\ c : \hat{\ }^{\mathtt{rw}}T$; and

- c is non-sendable in LP and $@_a(c!v \mid *c?p \to P)$,

we have:

$$\begin{aligned}&\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid @_a(c!v \mid *c?p \to P))\\ &\quad \succeq_\Gamma\ \texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid @_a(\mathsf{match}(p,v)P \mid *c?p \to P)).\end{aligned}$$

Proof: Standard. We first claim that: if $\Gamma, c : \texttt{\^{}rw}T \vdash @_a(c!v|*c?p \to P)$ then

$$\begin{aligned}&\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ @_a(c!v \mid *c?p \to P)\\ &\quad \succeq_\Gamma\ \texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ @_a(\mathsf{match}(p,v)P \mid *c?p \to P)\end{aligned}$$

This claim can be easily validated; we omit the details. Given the premises, we may derive the following.

$$\begin{aligned}&\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid @_a(c!v \mid *c?p \to P))\\ &\quad \succeq_\Gamma\ (\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid @_a *c?p \to P)) \mid (\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ @_a(c!v \mid *c?p \to P))\\ &\qquad \text{by Lemma 6.2.4 below}\\ &\quad \succeq_\Gamma\ \texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid @_a *c?p \to P)\\ &\qquad\quad \mid \texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ @_a(\mathsf{match}(p,v)P \mid *c?p \to P)\\ &\qquad \text{by the claim and Theorem 5.4.3}\\ &\quad \succeq_\Gamma\ \texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid @_a(\mathsf{match}(p,v)P \mid *c?p \to P))\\ &\qquad \text{by Lemma 6.2.4 below}\end{aligned}$$

Hence the lemma. ■

The proof of this relies on Lemma 6.2.4 below which can be informally described as follows. Imagine that two processes LP, LQ share a replicated resource $LR = @_a *c?p \to R$; moreover, such a resource is private, ie. the scope of c is over LP, LQ, LR. Provided that LP and LQ never use c for interacting between them, it makes no difference if we give LP and LQ its own private resource. The formal statement is given below.

Lemma 6.2.4 (Replicated resource)
Given a closed located type context Γ with

- $\Gamma,\ c : \texttt{\^{}rw}T \vdash LP|LQ$ and $(\Gamma,\ c : \texttt{\^{}rw}T, \Delta) \vdash_a R$, where $(\Gamma,\ c : \texttt{\^{}rw}T) \vdash p \in T \triangleright \Delta$;
- c is output-only in LP and LQ w.r.t. $\Gamma,\ c : \texttt{\^{}rw}T$, and in R w.r.t. $\Gamma,\ c : \texttt{\^{}rw}T, \Delta$; and
- c is non-sendable in LP, LQ, R,

we have:

$$\begin{aligned}&\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid LQ \mid @_a *c?p \to R)\\ &\quad \sim_\Gamma\ (\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LP \mid @_a *c?p \to R)) \mid (\texttt{new}\ c : \texttt{\^{}rw}T\ \texttt{in}\ (LQ \mid @_a *c?p \to R)).\end{aligned}$$

Proof: Standard. First we construct an indexed relation $\mathcal{S}$ as follows:

$$\mathcal{S}_{\Gamma'} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathbf{new}\ \Theta,\ c : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ (LP' \mid LQ' \mid @_a{*}c?p \to R), \\ \mathbf{new}\ \Theta,\ c, c' : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ ((LP' \mid @_a{*}c?p \to R) \\ \qquad \mid \{c'/c\}(LQ' \mid @_a{*}c?p \to R)) \end{array} \middle| \ \text{side-conditions} \right\}$$

where the side-conditions are:

- $\Gamma', \Theta,\ c : \hat{}^{\texttt{rw}}T \vdash LP'|LQ'$ and $\Gamma', \Theta,\ c : \hat{}^{\texttt{rw}}T, \Xi \vdash_a R$, where $\Gamma', \Theta,\ c : \hat{}^{\texttt{rw}}T \vdash p \in T \rhd \Xi$;
- $\vdash \Gamma', \Theta,\ c : \hat{}^{\texttt{rw}}T,\ c' : \hat{}^{\texttt{rw}}T$;
- c is non-sendable in LP', LQ' and R;
- c is output-only in LP' and LQ' w.r.t. $\Gamma', \Theta,\ c : \hat{}^{\texttt{rw}}T$; and
- c is output-only in R w.r.t. $\Gamma', \Theta,\ c : \hat{}^{\texttt{rw}}T, \Xi$.

We may then show that $\mathcal{S}$ a strong congruence (detail omitted).

Having proved such a result, we may now derive the lemma.

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{new}\ c : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ (LP \mid LQ \mid @_a{*}c?p \to R)} \\ \sim_\Gamma & \mathbf{new}\ c, c' : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ (LP \mid @_a{*}c?p \to R \mid \{c'/c\}(LQ \mid @_a{*}c?p \to R)) \\ \equiv & (\mathbf{new}\ c : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ (LP \mid @_a{*}c?p \to R)) \\ & \mid (\mathbf{new}\ c' : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ \{c'/c\}(LQ \mid @_a{*}c?p \to R)) \\ \equiv & (\mathbf{new}\ c : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ (LP \mid @_a{*}c?p \to R)) \\ & \mid (\mathbf{new}\ c : \hat{}^{\texttt{rw}}T\ \mathbf{in}\ (LQ \mid @_a{*}c?p \to R)) \end{array}$$

■

The above lemma is adapted from "the replication theorem" [Mil93b], adding location annotation. It expresses a useful distributivity property of private replicated processes and is essential in the proof of validity of β-reduction of encodings of the λ-calculus [Mil92], as well as in the proof of representability of Higher-Order π-calculus in first-order π-calculus [San93a]. Subsequent typing technology such as I/O-subtyping [PS96] reduces the original heavy side-condition, and uniform receptiveness [San99] enables the theorem to be used in the situation where the set of clients of the replicated resource may change dynamically. In the next subsection, we show how the latter can be adapted so as to deal with the situation where the agents in which replicated resources reside can be created dynamically.

6.2.3 Uniform Receptiveness

The model of functional computation in the previous section does not deal with the cases where function definitions (ie. replicated resources) reside in dynamically created agents. Consider the process below, for example.

$$@_a(\ast c\texttt{?}p\to P \mid \texttt{create}^{\texttt{m}}\ b = \ast c\texttt{?}p\to Q\ \texttt{in}\ (\langle b\rangle c\texttt{!}v|R))$$

Clearly, the output $\langle b\rangle c\texttt{!}v$ may react with the replicated input $\ast c\texttt{?}p\to Q$ in the newly created agent b; moreover, since the channel c is functional for agent b, such a reaction step will induce an expansion. This expansion cannot be derived using Lemma 6.2.3, however, since c is used for input both in $\ast c\texttt{?}p\to P$ and in the $\texttt{create}^{\texttt{m}}$ process.

To deal with the above situation, which is the case for `deliver` channel in the example infrastructure, we turn to Sangiorgi's notion of *uniform receptiveness* [San99]. A channel x is receptive in a process P if P is always ready to accept an input at x (at least as long as there are processes that could send messages at x). Uniformity of reception means that all inputs at x have the same continuation, implying the channel x is functional. We adapt this to a distributed setting, regarding x as uniformly receptive in LP if each agent in LP, including those which are dynamically created and **new**-bound, either contain a unique replicated input on x or does not use x for input.

We define two forms of judgements: $x : m \Vdash_a P$ and $x : S \Vdash LP$, where $m \in \{0, \omega\}$ and S is a set of names. Informally, $x : 0 \Vdash_a P$ means that x is never used for input in P, whereas $x : \omega \Vdash_a P$ means that P contains exactly one replicated input on x. The meta-variable m is therefore a *receptive variable* which is either 0 or ω. For located processes, $x : S \Vdash LP$ implies x is uniformly receptive in LP; moreover, each agent $a \in S$ contains a unique replicated input on x, whereas each agent $b \notin S$ does not use x for input. When $x : m \Vdash_a P$ and $x : S \Vdash LP$, the name x can be used for output in P and LP, but x is not sendable in such processes. This ensures that uniform receptiveness is preserved by the labelled transitions. Note that dynamically created and **new**-bound agents are not obliged to contain a function definition for x, but if they do then such a definition will be unique for such agents. This allows x to be uniformly receptive in agents created by the syntactic sugar eg. in $\langle b@s\rangle c\texttt{!}v$. The formal definition of uniform receptiveness is given below.

Definition 6.2.3 (Uniform Receptiveness)
Given that $\Gamma \vdash LP$ and $\Gamma \vdash x \in \hat{\ }^{\texttt{rw}}T$, the channel x is said to be *S-uniformly receptive* in LP if, $S \subseteq \mathsf{agents}(\Gamma)$ and $x : S \Vdash LP$, derivable by the rules given in Figures 6.2-6.3.

(UR-Nil)

$$x : 0 \Vvdash_a \mathbf{0}$$

(UR-Rep)

$$\frac{x : 0 \Vvdash_a P}{x : \omega \Vvdash_a \texttt{*}x\texttt{?}p \rightarrow P}$$

(UR-New)

$$\frac{x : m \Vvdash_a P \quad x \notin \mathsf{dom}(\Delta)}{x : m \Vvdash_a \texttt{new}\ \Delta\ \texttt{in}\ P}$$

(UR-IfLocal)

$$\frac{x : 0 \Vvdash_a P,\ Q \quad x \notin \mathsf{fv}(v)}{x : 0 \Vvdash_a \texttt{iflocal}\ \langle b\rangle x\texttt{!}v\ \texttt{then}\ P\ \texttt{else}\ Q}$$

(UR-Create)

$$\frac{x : m \Vvdash_b P \quad x : 0 \Vvdash_a Q}{x : 0 \Vvdash_a \texttt{create}^Z\ b = P\ \texttt{in}\ Q}$$

(UR-Plain)

$$\frac{x : 0 \Vvdash_a P,\ Q \quad c \neq x \quad x \notin \mathsf{fv}(v) \quad x \notin \mathsf{fv}(ev)}{\begin{array}{ll} x : 0 \Vvdash_a & \texttt{let}\ p = ev\ \texttt{in}\ P,\ \texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ Q, \\ & \texttt{iflocal}\ \langle b\rangle c\texttt{!}v\ \texttt{then}\ P\ \texttt{else}\ Q,\ \texttt{migrate to}\ s \rightarrow P, \\ & c\texttt{?}p \rightarrow P,\ \texttt{*}c\texttt{?}p \rightarrow P,\ c\texttt{!}v,\ \langle b\rangle c\texttt{!}v,\ \langle b@s\rangle c\texttt{!}v,\ \langle b@?\rangle c\texttt{!}v \end{array}}$$

(UR-Out)

$$\frac{x \notin \mathsf{fv}(v)}{x : 0 \Vvdash_a x\texttt{!}v,\ \langle b\rangle x\texttt{!}v,\ \langle b@s\rangle x\texttt{!}v,\ \langle b@?\rangle x\texttt{!}v}$$

(UR-Par)

$$\frac{x : m_1 \Vvdash_a P \quad x : m_2 \Vvdash_a Q \quad (m_1 = 0) \vee (m_2 = 0)}{x : m_1 + m_2 \Vvdash_a P \mid Q}$$

Figure 6.2: Uniform receptiveness: Basic process

(UR-At)

$$\frac{z \in S \Rightarrow x : \omega \Vvdash_z P \quad z \notin S \Rightarrow x : 0 \Vvdash_z P}{x : S \Vvdash @_z P}$$

(UR-LPar)

$$\frac{x : S_1 \Vvdash LP \quad x : S_2 \Vvdash LQ \quad S_1 \cap S_2 = \emptyset}{x : S_1 \cup S_2 \Vvdash LP \mid LQ}$$

(UR-LNew)

$$\frac{\exists S' \subseteq \mathsf{agents}(\Delta)\ .\ x : S \cup S' \Vvdash LP \quad x \notin \mathsf{dom}(\Delta) \quad S \cap \mathsf{dom}(\Delta) = \emptyset}{x : S \Vvdash \texttt{new}\ \Delta\ \texttt{in}\ LP}$$

Figure 6.3: Uniform receptiveness: Located process

We prove a result similar to Lemma 6.2.3, ie. that a τ-step caused by a communication on a uniformly-receptive channel gives rise to an expansion. The proof of such a lemma relies on the preservation of uniform receptiveness under transitions.

Lemma 6.2.5 (Uniform receptiveness is preserved by LTS)
Given that Γ is a closed located type context with $\Gamma \vdash LP$, if $x : S \Vdash LP$ with $S \subseteq \mathsf{agents}(\Gamma)$ and $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then the following hold:

- if β is an output label then there exists $S' \subseteq \mathsf{agents}(\Delta)$ such that $x : S \cup S' \Vdash LQ$;
- if $\beta = @_a c?v$, for some a, c, v, and $x \notin \mathsf{fv}()$ then $x : S \Vdash LQ$;
- if α is a migrate or τ label then then $x : S \Vdash LQ$.

Proof: An induction on the derivation of $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$. ■

Lemma 6.2.6 (Functional computation yields an expansion)
Given a closed located type context Γ, supposing the following hold:

- $\Gamma \vdash \mathbf{new}\ c : \hat{}^{\mathtt{rw}}T\ \mathbf{in}\ (LP \mid @_a(c!v \mid *c?p \to P))$.
- For some $S \subseteq \mathsf{agents}(\Gamma)$, $c : S \Vdash LP \mid @_a(c!v \mid *c?p \to P)$.

then

$$\begin{aligned}&\mathbf{new}\ c : \hat{}^{\mathtt{rw}}T\ \mathbf{in}\ (LP \mid @_a(c!v \mid *c?p \to P))\\ &\quad \succeq_\Gamma\ \mathbf{new}\ c : \hat{}^{\mathtt{rw}}T\ \mathbf{in}\ (LP \mid @_a(\mathsf{match}(p,v)P \mid *c?p \to P)).\end{aligned}$$

Proof: In order to prove this, we construct the following relation.

$$\mathcal{R}_\Phi \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \mathbf{new}\ \Theta\ \mathbf{in}\ (LQ \mid @_a(c!v \mid *c?p \to P)), \\ \mathbf{new}\ \Theta\ \mathbf{in}\ (LQ \mid @_a(\mathsf{match}(p,v)P \mid *c?p \to P)) \end{array} \;\middle|\; \text{side-condition} \right\} \cup \equiv_\Phi$$

where the side condition is: there exists $S \subseteq \mathsf{agents}(\Phi, \Theta)$ such that

- $\Phi \vdash \mathbf{new}\ \Theta\ \mathbf{in}\ (LQ \mid @_a(c!v \mid *c?p \to P))$;
- $c : S \Vdash LQ \mid @_a(c!v \mid *c?p \to P)$; and
- $a \in \mathsf{dom}(\Phi)$ and $c \in \mathsf{dom}(\Theta)$.

We may then prove that $\mathcal{R}$ is a translocating expansion congruence. We omit the details. ■

Amadio et al. [ABL99] also adapted the concept of uniform receptiveness to a distributed setting. Their work is different from ours in many respects:

- their type system ensures that *all* channels are uniformly receptive;

- as in the distributed join-calculus, for each channel c there exists a unique location in which c can be used for input; and
- inter-agent messages are guaranteed to find appropriate receptors at their target locations.

These restrictions require a new programming style, many examples of which are given by the authors.

6.2.4 Local Channels

In distributed process calculi literature, a channel is said to have a *locality* if there exists a unique location which can use such a channel either for input, for output, or for both. In the distributed join-calculus [FGL+96], for each channel there exists a unique location where inputs on such a channel reside (cf. Section 8.1 on page 156). More generally, Sewell [Sew98] studied a type system which refines channel types with global and local capabilities as well as input and output. Using such a type system, the join-calculus channels, for example, can be thought of as roughly those which have global output and local input capabilities. A different view of locality is given by Amadio and Prasad [AP94]. They studied a calculus in which each channel is associated with a location in such a way that if a location fails, then all the channel associated with such a location can no longer be used for communication.

In this section, we study channels which are *local* in the sense that they are never used for inter-agent communication; an example of this is `currentloc` in the $\mathcal{C}$-translation. We formally define this below.

Definition 6.2.4 (Local Channels)
A channel c is said to be *local* in LP if it is non-sendable in LP and $c \notin \mathsf{chSubj}(LP)$, where $\mathsf{chSubj}(LP)$ is the set of channels which are used for output in **`iflocal`** or LI output primitives in LP.

The exact definition of $\mathsf{chSubj}(\cdot)$ can be obtained from that of $\mathsf{chObj}(\cdot)$, given in Figure 6.1, by replacing the rules involving outputs with the following.

$$
\begin{aligned}
\mathsf{chSubj}(c\,!\,v) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{chSubj}(\langle b@?\rangle c\,!\,v) &\stackrel{\text{def}}{=} \{c\} \\
\mathsf{chSubj}(\texttt{\textbf{iflocal}}\ \langle b\rangle c\,!\,v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q) &\stackrel{\text{def}}{=} \{c\} \cup \mathsf{chSubj}(P) \cup \mathsf{chSubj}(Q)
\end{aligned}
$$

Consider the example process below.

$$\texttt{\textbf{new}}\ c : \texttt{\^{}}^{\texttt{rw}}T\ \texttt{\textbf{in}}\ (@_a P \mid @_b Q)$$

If a channel c is used locally in $(@_a P \mid @_b Q)$ then the agent a will not use c for sending messages to b, or vice versa. This means that we can, without changing the observable behaviours, redefine the scope of c so it is "localised" to each agent. We generalise this intuition to arbitrary located processes below.

Lemma 6.2.7 (Scope narrowing of local channel)
Given that Γ is a closed located type context with $\Gamma,\ c : \texttt{\^{}}^{I}T \vdash LP|LQ$, if c is a local channel in LP, LQ and $\mathsf{agents}(LP) \cap \mathsf{agents}(LQ) = \emptyset$ then

$$\texttt{\textbf{new}}\ c : \texttt{\^{}}^{I}T\ \texttt{\textbf{in}}\ (LP \mid LQ) \quad \sim_\Gamma \quad (\texttt{\textbf{new}}\ c : \texttt{\^{}}^{I}T\ \texttt{\textbf{in}}\ LP) \mid (\texttt{\textbf{new}}\ c : \texttt{\^{}}^{I}T\ \texttt{\textbf{in}}\ LQ).$$

where $\mathsf{agents}(LP)$ is the set of agents in LP which syntactically locate a basic process.

$$\begin{array}{rcl}
\mathsf{agents}(@_a P) & \stackrel{\text{def}}{=} & \{a\} \\
\mathsf{agents}(LP|LQ) & \stackrel{\text{def}}{=} & \mathsf{agents}(LP) \cup \mathsf{agents}(LQ) \\
\mathsf{agents}(\texttt{\textbf{new}}\ \Delta\ \texttt{\textbf{in}}\ LP) & \stackrel{\text{def}}{=} & \mathsf{agents}(LP)/\mathsf{dom}(\Delta)
\end{array}$$

Proof Sketch: It is awkward to construct a translocating congruence using the premises of this lemma. In particular, we need to explicitly rearrange processes after transition so that the premise $\mathsf{agents}(LP) \cap \mathsf{agents}(LQ) = \emptyset$ holds (consider eg. inter-agent messaging between LP and LQ). It is simpler to ensure that c will not used for a communication between LP and LQ. For this we define $\mathsf{writeA}(c, LP)$ and $\mathsf{readA}(c, LP)$ as the sets of agents which may use c for reading and writing respectively.

$$\begin{array}{lcl}
\mathsf{readA}_a(c, c\texttt{?}p \to P) & \stackrel{\text{def}}{=} & \{a\} \\
\mathsf{readA}_a(c, \texttt{*}c\texttt{?}p \to P) & \stackrel{\text{def}}{=} & \{a\} \\
\mathsf{readA}(c, @_a P) & \stackrel{\text{def}}{=} & \mathsf{readA}_a(c, P) \\
 & & \\
\mathsf{writeA}_a(c, c\texttt{!}v) & \stackrel{\text{def}}{=} & \{a\} \\
\mathsf{writeA}_a(c, \texttt{\textbf{iflocal}}\ \langle b\rangle c\texttt{!}v\ \texttt{\textbf{then}}\ P\ \texttt{\textbf{else}}\ Q) & \stackrel{\text{def}}{=} & \{a\} \\
\mathsf{writeA}(c, @_a P) & \stackrel{\text{def}}{=} & \mathsf{writeA}_a(c, P)
\end{array}$$

For the rest, $\mathsf{readA}(c, \cdot)$ and $\mathsf{writeA}(c, \cdot)$ are homomorphic. We observe that if c is a local channel then the sets $\mathsf{writeA}(c, LP)$ and $\mathsf{readA}(c, LP)$ are preserved by transitions in which

c is not received as an argument. The problematic premise can be replaced by the following, which is preserved by transitions.

$$\mathsf{readA}(c, LP) \cap \mathsf{writeA}(c, LQ) = \mathsf{readA}(c, LQ) \cap \mathsf{writeA}(c, LP) = \emptyset$$

We may now construct a translocating indexed relation, and prove that it is a translocating strong bisimulation. The details are given in Appendix B.8 on page 221. ■

Readjusting the scopes of names is necessary for the uses and the proofs of temporary immobility later. For example, it is simpler to show that an agent a encapsulated by

$$\mathbf{new}\ \mathtt{currentloc}\ \mathbf{in}\ @_a(\llbracket P \rrbracket \ \mid \mathtt{ack?}[] \rightarrow ..)$$

is temporarily immobile blocked by `ack`, than to deal with the whole system of many agents, bound with `currentloc` at the top-level.

6.3 Determinacy and Confluence

Components in a concurrent program are often independent, in the sense that execution of one component does not (immediately) affect that of the others. This is reflected in operational semantics when transitions originated by different components commute. The notion of *confluence*, used in process calculi by Milner in [Mil80, Mil89], formally captures this phenomenon. For clarity, we recapitulate the definition of strong confluence for CCS processes below. Note that in this definition $\dot\sim$ is the strong bisimulation defined for CCS processes, and a, b are CCS actions.

Definition 6.3.1 (Strong confluence *à la* [Mil80])
P is always strongly 0-confluent.
P is strongly $(k+1)$-confluent iff:

1. $P \xrightarrow{a} P_1$ and $P \xrightarrow{b} P_2$ implies either:
   - $a = b$ and $P_1 \dot\sim P_2$ or
   - there exists P_3 and P_4 such that $P_1 \xrightarrow{b} P_3$, $P_2 \xrightarrow{a} P_4$ and $P_3 \dot\sim P_4$.
2. $P \xrightarrow{\mu} Q$ implies Q strongly k-confluent.

P is strongly confluent iff it is strongly k-confluent for all $k \geq 0$.

The key property of a confluent system is that it implies τ*-inertness* (or τ-stability in [Tof91]), where a process P is τ-inert if it satisfies:

$$\text{If } P \xrightarrow{\tau} Q \text{ then } P \mathrel{\dot{\succeq}} Q$$

where here $\dot{\succeq}$ is the expansion defined for CCS processes. Note the similarity to sequential computation where reduction does not change the value of a term: here (internal) reduction does not change a term's behaviour. This property can be very useful for process verification since it allows the state space to be reduced, sometimes drastically, by performing as many internal reductions as possible without changing the equivalence class of the process under investigation (see some examples in [GS95]).

Realistic protocols do not often meet the requirement set by Definition 6.3.1, however. Alternative definitions, such as progressing confluence [GS95] and partial confluence [Phi96], have been formulated. These definitions retain the τ-inertness property, but relax the clauses of Definition 6.3.1 so that only certain actions are required to commute with all other actions. Nevertheless such notions of confluence involve quantification over all derivatives, and can therefore be difficult to establish. Moreover, compositional techniques cannot be applied as confluence is not preserved by parallel composition (unless composed processes share no free names and cannot communicate with each other). Yet a notion of confluence is useful for proving our infrastructure correct, since many of the additional reductions introduced by the encoding are essentially confluent.

When proving the example infrastructure correct, we observe that a confluent step is often introduced by a subprocess which is τ*-deterministic*. A process is said to be τ-deterministic if, under a translocation, the next computational step of a process is completely determined. For example, a conditional **if true then** P **else** Q reduces to P regardless of the location context, and $@_a\langle b@s\rangle c\mathtt{!}v$ reduces to $@_b c\mathtt{!}v$ provided that b is located at s and is not to be moved. A key property of a τ-deterministic process is that its reduction step induces a translocating expansion. Hence, if placed in a program context which respects its translocation, a reduction generated from a τ-deterministic process is τ-inert. The formal definition of τ-determinism is given below.

Definition 6.3.2 (Deterministic reduction)
Given a closed located type context Γ and $M \subseteq \mathsf{mov}(\Gamma)$, a located processes LP is said to *deterministically reduce* to LQ w.r.t. (Γ, M), written $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$, if, for any valid δ for (Γ, M), the following hold:

- $\Gamma\delta \Vdash LP \xrightarrow{\tau} LQ$; and

- $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LQ'$ implies $\beta = \tau$, $\Delta = \bullet$ and $LQ' \dot{\sim}^{M}_{\Gamma\delta} LQ$.

We also define the relation $\overset{\mathsf{det}}{\underset{M}{\Longrightarrow}}$ to be the transitive closure of $\xrightarrow[M]{\mathsf{det}}$; that is $\Gamma \Vdash LP \overset{\mathsf{det}}{\underset{M}{\Longrightarrow}} LQ$ implies there exists $LP_1, \ldots, LP_n$ such that, letting $LP = LP_0$ and $LQ = LP_{n+1}$, we have

$$\Gamma \Vdash LP_i \quad \xrightarrow[M]{\mathsf{det}} \quad LP_{i+1} \qquad 0 \leq i \leq n$$

A process LP is said to be τ-deterministic w.r.t. Γ, M if there exists LQ such that $\Gamma \Vdash LP \overset{\mathsf{det}}{\underset{M}{\Longrightarrow}} LQ$.

A non-trivial example of a τ-deterministic process is the following.

$$\begin{array}{l}\textbf{new}\ m : \mathtt{Map[Agent^s\ Site]},\ \mathtt{dack} : \mathtt{\hat{}^{rw}}[]\ \textbf{in}\\ \quad (@_D\langle a@s\rangle\mathtt{deliver!}\ \{T\}\,[c\ v\ \mathtt{dack}]\\ \quad \mid @_D(\mathsf{makeMap}(m;map) \mid \mathtt{dack?}[] \rightarrow \mathtt{lock!}m))\end{array}$$

This is a part of the daemon which is about to forward a message $c!v$ to an agent a at site s. Note that the input on `dack` cannot receive a message until the LD output has extruded the scope of `dack`.

The lemma below states the key property of τ-determinacy: that a deterministic reduction gives rise to a translocating expansion.

Lemma 6.3.1 (Deterministic reduction induces expansion)
If $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$ then $LP \dot{\succeq}^{M}_{\Gamma} LQ$.

Proof: Construct a translocating indexed relation $\mathcal{S}$ such that

$$\mathcal{S}^{M'}_{\Phi} \overset{\mathrm{def}}{=} \{(LP, LQ) \mid \Phi \Vdash LP \xrightarrow[M']{\mathsf{det}} LQ\} \cup \dot{\sim}^{M'}_{\Phi}$$

This can be easily proved to be a translocating expansion. ∎

Many internal computations, including LD-messaging, are τ-deterministic (Facts 6.3.2 to 6.3.6 below). Fact 6.3.7 shows the confluence of communication along *linear* channels, ie. those that can be used at most once throughout their lifetime for communication. This result is simple compared to that of [KPT96], which uses a refined type system, yet it is sufficient for proving τ-inertness of communication via acknowledgement channels in the example infrastructure.

Lemma 6.3.2 (Determinacy of conditional statement)
Given that Γ is a closed located type context, and $\Gamma \vdash_a P, Q$. For all $M \subseteq \mathsf{agents}(\Gamma)$, the following hold.

- $\Gamma \Vdash @_a\mathtt{if\ true\ then}\ P\ \mathtt{else}\ Q \xrightarrow[M]{\mathsf{det}} @_a P$.
- $\Gamma \Vdash @_a\mathtt{if\ false\ then}\ P\ \mathtt{else}\ Q \xrightarrow[M]{\mathsf{det}} @_a Q$.

Lemma 6.3.3 (Determinacy of local messaging)
Given that Γ is a closed located type context, and $\Gamma \vdash_a \mathtt{iflocal}\ \langle b\rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ Q$. For all $M \subseteq \mathsf{agents}(\Gamma)/\{a, b\}$, the following hold.

- $\Gamma \vdash a@s$ and $\Gamma \vdash b@s$ implies:

$$\Gamma \Vdash @_a\mathtt{iflocal}\ \langle b\rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow[M]{\mathsf{det}} @_a P \mid @_b c!v.$$

- $\Gamma \vdash a@s$, $\Gamma \vdash b@s'$ with $s \neq s'$ implies:

$$\Gamma \Vdash @_a\mathtt{iflocal}\ \langle b\rangle c!v\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow[M]{\mathsf{det}} @_a Q.$$

Lemma 6.3.4 (Determinacy of agent creation)
Given that Γ is a closed located type context, and $\Gamma \vdash_a \mathtt{create}^Z\ b = P\ \mathtt{in}\ Q$. If $\Gamma \vdash a@s$ then, for all $M \subseteq \mathsf{agents}(\Gamma)/\{a\}$, the following holds.

$$\Gamma \Vdash @_a\mathtt{create}^Z\ b = P\ \mathtt{in}\ Q \quad \xrightarrow[M]{\mathsf{det}} \quad \mathtt{new}\ b : \mathtt{Agent}^Z@s\ \mathtt{in}\ (@_b P | @_a Q)$$

Lemma 6.3.5 (Determinacy of inter-agent messaging)
Given that Γ is a closed located type context, $\Gamma \vdash_a \langle b@s\rangle c!v$ and $\Gamma \vdash b@s$. For any $M \subseteq \mathsf{agents}(\Gamma)/\{b\}$, we have:

$$\Gamma \Vdash @_a\langle b@s\rangle c!v \quad \overset{\mathsf{det}}{\underset{M}{\Longrightarrow}} \quad @_b c!v.$$

Lemma 6.3.6 (Determinacy of evaluation)
Given that Γ is a closed located type context, and $\Gamma \vdash @_a\mathtt{let}\ p = ev\ \mathtt{in}\ P$. For any $M \subseteq \mathsf{agents}(\Gamma)$, we have:

$$\Gamma \Vdash @_a\mathtt{let}\ p = ev\ \mathtt{in}\ P \quad \xrightarrow[M]{\mathsf{det}} \quad @_a\mathsf{match}(p, \mathtt{eval}(ev))P.$$

Lemma 6.3.7 (Determinacy of use-once channel communication)
Given thatΓ is a closed located type context, and $\Gamma,\ c : \hat{\ }^{\mathtt{rw}}T \vdash @_a(c!v \mid c?p \to P)$. For any $M \subseteq \mathsf{agents}(\Gamma)$, we have:

$$\Gamma \Vdash @_a\mathtt{new}\ c : \hat{\ }^{\mathtt{rw}}T\ \mathtt{in}\ (c!v \mid c?p \to P) \quad \xrightarrow[M]{\mathsf{det}} \quad @_a\mathtt{new}\ c : \hat{\ }^{\mathtt{rw}}T\ \mathtt{in}\ \mathsf{match}(p, v)P.$$

The theory of determinacy and confluence for the π-calculus has been an important verification tool. It has been employed for protocol verification [Mil89, GS95], on-the-fly reduction of finite state spaces [HP94, Qin91], and reasoning about objects [Phi96]. By comparison, our notion of τ-determinacy is restrictive, since the deterministic reductions do not commute with some migrate actions. Nevertheless we can use mutual exclusion techniques for ensuring that such migrate actions will not occur before the deterministic reduction steps. For example, a deterministic reduction may subsequently release a lock which an agent must acquire in order to migrate. In Section 6.4 we make this precise, and prove that in such situations a reduction by a τ-deterministic component is τ-inert.

6.4 Temporary Immobility

It is generally difficult to guarantee the safe delivery of location-dependent messages to mobile agents. In practice, some means of mutual exclusion is often used to ensure that, while an LD operation is being executed, the agents involved are not migrating. Our example infrastructure algorithm (as well as those presented in [SWP99, Woj00a]) uses *lock* channels (both within the daemon and each agent) to ensure such mutual exclusions. The daemon lock ensures that whenever an agent migrates, the site map of the daemon is updated accordingly before the daemon deals with the next request. This section makes precise such informal reasoning. It identifies processes which are *temporarily immobile*, waiting for a lock to be released or an acknowledgement from the daemon. This technique is vital in infrastructure verification, since it guarantees the safety of LD communication between the daemon and agents.

Intuitively, an agent a in a process LP is prevented from migrating if there exists an input (or output) action which *always* precedes migration of a in any derivative of LP. To formalise this, we need to determine the possible sequences of actions of a located process w.r.t. a located context. However, to simplify the task of evaluating all possible paths, we may decompose the process into smaller subprocesses and only concentrate on the essential part. This means we must introduce a translocating index for handling possible relocations of agents by program contexts. For this, we define translocating paths below.

Definition 6.4.1 (Translocating Paths)
A *translocating path* of LP_0 w.r.t. (Γ, M) is a sequence

$$\xrightarrow[\Delta_1]{\beta_1} \cdots \xrightarrow[\Delta_n]{\beta_n}$$

for which there exist $LP_1, \ldots, LP_n$ and $\delta_0, \ldots, \delta_{n-1}$ such that for each $i \in 0 \ldots n-1$:

- δ_i is a relocator, valid w.r.t. $\Gamma, \Delta_1, \ldots, \Delta_i$;
- $\mathsf{dom}(\delta_i) \subseteq M \uplus_{\beta_1} \mathsf{mov}(\Delta_1) \ldots \uplus_{\beta_i} \mathsf{mov}(\Delta_i)$; and
- $((\Gamma\delta_0,\ \Delta_1)\delta_1\beta_1,\ \Delta_2 \ldots \beta_i,\ \Delta_i)\delta_i \Vdash LP_i \xrightarrow[\Delta_{i+1}]{\beta_{i+1}} LP_{i+1}$.

$LP_1, \ldots, LP_n$ are then said to be *translocating derivatives* of LP with respect to Γ under M.

There are many ways in which the intuition about temporary immobility can be formalised. Here we choose a notion which is applicable to the example infrastructure and is relatively easy to establish: a located process LP is said to be temporarily immobile if no migration of any agent can occur unless preceded by an input on the lock channel. To ease the task of proving temporary immobility, we cut down the space of translocating derivatives required to be checked by assuming that the lock channel is non-sendable, and hence never received from the environment as an argument. To ensure this, we also insist that the lock channel is non-sendable in the temporarily immobile process. The formal definition is given below:

Definition 6.4.2 (Temporary Immobility)
Given a closed located type context Γ, a located process LP with $\Gamma \vdash LP$, and a translocating index $M \subseteq \mathsf{agents}(\Gamma)$, LP is *temporarily immobile* under lock l w.r.t. (Γ, M) if, for all translocating paths

$$\xrightarrow[\Delta_1]{\beta_1} \cdots \xrightarrow[\Delta_n]{\beta_n}$$

of LP w.r.t. (Γ, M) which do not contain an input action $\beta_j = @_a c\boldsymbol{?}v$ with $l \in \mathsf{fv}(c, v)$, the following hold for all $i \leq n$, b, c, v and s:

- $\beta_i = @_b c\boldsymbol{!}v$ implies $l \notin \mathsf{fv}(\beta_i)$; and
- $\beta_i \neq @_b\mathsf{migrate\ to}\ s$.

Note that a temporarily immobile process LP may contain parts of many agents. The above definition, however, ignores the location annotation of the input on the lock channel action, as well as that of the migrate action. This allows the following situations to be dealt with:

- More than one agent may be prevented from migrating, until the lock is released. This is certainly the case for the example infrastructure algorithm; for example, neither a newly-created agent nor its parent can migrate until the former receives an acknowledgement from the daemon.
- Receiving an input on the lock channel on one agent may allow *another* agent to migrate. This situation is illustrated by the example process in (6.1) below, where an agent cannot migrate until the daemon sucessfully acquires the daemon lock `lock`.

Definition 6.4.2 captures our intuition of temporary immobility. Proving that a process is temporarily immobile, however, can be difficult for we need to perform a case analysis over possible sequences of transitions. A coinductive proof technique is preferable for proving temporary immobility — not only as such a technique allows analysis of single step transitions, but also as it allows the space of possible transition sequences to be cut down by using an "up to" technique (see later). Following this, we formulate *blocking sets.* A blocking set is a set of located processes (indexed by a translocating index and a located type context) which is closed under transitions that are not input actions containing l; moreover, none of the processes in such a set can immediately migrate. The formal definition is as follows.

Definition 6.4.3 (Blocking Set)
A translocating unary relation $\mathcal{M}$ is a blocking set under lock l if, for any $LQ \in \mathcal{M}_{\Gamma}^{M}$, $M \subseteq \mathsf{mov}(\Gamma)$, $\Gamma \vdash LQ$, and valid relocator δ for (Γ, M), whenever $\Gamma\delta \Vdash LQ \xrightarrow[\Theta]{\beta} LQ'$, we have:

- $\beta = \tau$ implies $LQ' \in \mathcal{M}_{\Gamma\delta}^{M}$;
- $\beta = @_b c\textbf{?}v$ with $l \notin \mathsf{fv}(\beta)$ implies $LQ' \in \mathcal{M}_{\Gamma\delta,\Theta}^{M\cup\mathsf{mov}(\Theta)}$;
- $\beta = @_b c\texttt{!}v$ implies $l \notin \mathsf{fv}(\beta)$ and $LQ' \in \mathcal{M}_{\Gamma\delta,\Theta}^{M}$; and
- $\beta \neq @_b\mathsf{migrate\ to}\ s$, for any b, s.

It is not difficult to check that if a process belongs to a blocking set under l then it is temporarily immobile under the same lock, and vice versa.

Lemma 6.4.1 (Coincidence of two temporary immobility definitions)
Let Γ be a closed located type context and $M \subseteq \mathsf{agents}(\Gamma)$ be a translocating index. LP is temporary immobile under l w.r.t. $(\Gamma,\ M)$ if and only if there exists a blocking set $\mathcal{M}$ under l such that $LP \in \mathcal{M}_{\Gamma}^{M}$.

To see how temporary immobility is used, consider the example process below.

$$\begin{aligned} LQ \stackrel{\text{def}}{=} \ & \textbf{new}\ \Omega_{aux}\ \textbf{in} \\ & \quad @_D Daemon \\ & \quad |\ @_a(\llbracket P \rrbracket_a | \texttt{currentloc!}s | \texttt{Deliverer}) \end{aligned} \qquad (6.1)$$

where $\Omega_{aux} = \Phi_{aux}/\Omega_D$ and

$$\begin{aligned} \Omega_D \stackrel{\text{def}}{=} \quad & D : \texttt{Agent}^{\mathsf{s}}@SD, \\ & \texttt{lock} : \texttt{\^{}}^{\texttt{rw}}\texttt{Map}[\texttt{Agent}^{\mathsf{s}}\ \texttt{Site}], \\ & \texttt{deliver} : \texttt{\^{}}^{\texttt{rw}}\ \{X\}\ [\texttt{\^{}}^{\texttt{w}}X\ X\ \texttt{\^{}}^{\texttt{w}}[]] \end{aligned}$$

Here agent a cannot migrate until the daemon lock `lock` is successfully acquired, so LQ is temporarily immobile under `lock` with respect to any type-correct (Γ, M) that does not admit environmental relocation of a, ie. with $a \notin M$. Assume further that a is at s and that the daemon is forwarding an LI message to a, ie. the above is in parallel with

$$LP \stackrel{\text{def}}{=} @_D\langle a@s\rangle\texttt{deliver!}[c\ v\ \texttt{ack}]$$

This parallel composition, with a surrounding new-binder for `lock`, expands to

$$\textbf{new}\ \texttt{lock} : \texttt{\^{}rw}\texttt{Map}[\texttt{Agent}^{\texttt{s}}\ \texttt{Site}]\ \textbf{in}\ (LQ \mid @_a\texttt{deliver!}[c\ v\ \texttt{ack}])$$

To prove this expansion, we consider the transitions of LP. Assuming that a remains at s, such transitions will eventually deliver the message to a at s, as shown below.

$$\begin{array}{lcl} LP & = & @_D\textbf{create}^{\texttt{m}}\ b = \textbf{migrate to}\ s \to (\textbf{iflocal}\ \langle a\rangle\texttt{deliver!}[c\ v\ \texttt{ack}]\ \textbf{then}\ \mathbf{0})\ \textbf{in}\ \mathbf{0} \\ & \xrightarrow{\tau} & \textbf{new}\ b : \texttt{Agent}^{\texttt{m}}@SD\ \textbf{in}\ @_b\textbf{migrate to}\ s \to (\textbf{iflocal}\ \langle a\rangle\texttt{deliver!}[c\ v\ \texttt{ack}]\ \textbf{then}\ \mathbf{0}) \\ & \xrightarrow{\tau} & \textbf{new}\ b : \texttt{Agent}^{\texttt{m}}@s\ \textbf{in}\ @_b\textbf{iflocal}\ \langle a\rangle\texttt{deliver!}[c\ v\ \texttt{ack}]\ \textbf{then}\ \mathbf{0} \\ & \xrightarrow{\tau} & @_a\texttt{deliver!}[c\ v\ \texttt{ack}] \end{array}$$

We observe that all these transitions are deterministic reduction. Moreover, since none of these steps yields an output on `lock` and LQ is temporarily immobile, blocked by `lock`, the agent a will remain at s while the above transitions occur. This can be generalised to the following lemma, capturing the key property of temporarily immobile processes.

Lemma 6.4.2 (Safety of deterministic reduction)
Given that LQ is temporary immobile under l w.r.t. $((\Gamma, \Delta), M)$, with Δ being extensible and $l \in \mathsf{dom}(\Delta)$, if $\Gamma, \Delta \Vdash LP_1 \xrightarrow[M]{\mathsf{det}} LP_2$ then

$$\textbf{new}\ \Delta\ \textbf{in}\ (LP_1 \mid LQ) \stackrel{\cdot}{\succeq}_{\Gamma}^{M \cap \mathsf{dom}(\Gamma)} \textbf{new}\ \Delta\ \textbf{in}\ (LP_2 \mid LQ)$$

Proof Sketch: The basic idea here is that LP_1 cannot trigger the lock, since it has a deterministic reduction to LP_2. As the lock is not triggered by LP_1, agents in LQ cannot migrate until after LP_1 safely reduces to LP_2.

Since LQ is temporarily immobile under l w.r.t. $((\Gamma, \Delta), M)$, there exists $\mathcal{M}$, a blocking set under lock l, such that $LQ \in \mathcal{M}^M_{\Gamma,\Delta}$. To prove this lemma, we construct a translocating indexed relation $\mathcal{R}$ as follows.

$$\mathcal{R}^{M'}_{\Xi} = \{\ (\textbf{new}\ \Delta\ \textbf{in}\ (LP_1 \mid LR),\ \textbf{new}\ \Delta\ \textbf{in}\ (LP_2 \mid LR)) \mid \text{side-condition}\ \} \cup\ \equiv_{\Xi}$$

where the side-condition is that there exists $\Gamma, \Theta_{in}, \Theta_{ex}, M''$ satisfying:

- $\Xi \equiv \Gamma, \Theta_{ex}, \Theta_{in}$;
- $\Xi, \Delta \vdash LP_1, LR$;
- $M' = (M'' \cap \mathsf{mov}(\Gamma)) \cup \mathsf{mov}(\Theta_{in})$;
- $l \in \mathsf{dom}(\Delta)$;
- $LR \in \mathcal{M}^{M'' \cup \mathsf{mov}(\Theta_{in})}_{\Xi,\Delta}$; and
- $\Xi, \Delta \Vdash LP_1 \xrightarrow[M'' \cup \mathsf{mov}(\Theta_{ex}, \Theta_{in})]{\mathsf{det}} LP_2$.

Informally, Θ_{in} is the located type context extruded from the environment by input actions, Θ_{ex} is that extruded to the environment by output actions of LR (LP_1 can never commit an output until it reduces to LP_2). The translocating index M' must include all mobile agents received from the environment, but it does not need to contain those of Θ_{ex}. The premises of the lemma which concern temporary immobility and deterministic reduction are extended, so that they deal with the new agents in Θ_{in} and Θ_{ex}.

This relation $\mathcal{R}$ is proved to be a translocating expansion. Details can be found in Appendix C on page 228. ■

Example 6.1 shows a situation in the $\mathcal{C}$-translation where an agent is temporarily immobile, blocked by the daemon lock. An agent can also be temporarily immobile if it is waiting for its local lock, for an acknowledgement from the daemon, or from a newly-created child. These situations are captured precisely in the following lemmas.

Lemma 6.4.3 (Blocked by daemon lock)
Given that Γ is a closed located type context, $\mathtt{lock} \notin \mathsf{fv}(P)$ and $M \subseteq \mathsf{agents}(\Gamma)/\{a\}$, the following processes (each well-typed w.r.t. Γ, Ω_D)

$$\begin{array}{l}
\textbf{new}\ \Omega_{aux}\ \textbf{in}\\
\quad @_D(\mathit{Daemon} \mid \prod_i \mathsf{mesgReq}\{T_i\}\ [a\ c_i\ v_i])\\
\quad \mid @_a([\![P]\!]_a \mid \mathtt{currentloc!}s \mid \mathtt{Deliverer})
\end{array}$$

$$\begin{array}{l}
\textbf{new}\ \Omega_{aux},\ \mathtt{rack} : \mathtt{\hat{}rw}[],\ \mathtt{pack} : \mathtt{\hat{}rw}[],\ b : \mathtt{Agent}^Z @ s\ \textbf{in}\\
\quad @_D(\mathit{Daemon} \mid \mathsf{regReq}[b\ s\ \mathtt{rack}] \mid \prod_i \mathsf{mesgReq}\{T_i\}\ [a\ c_i\ v_i])\\
\quad \mid @_a(\mathsf{regBlockP}(\mathtt{pack}\ Q) \mid [\![P]\!]_a \mid \mathtt{Deliverer}) \mid @_b \mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ R)
\end{array}$$

$$\begin{array}{l}
\textbf{new}\ \Omega_{aux},\ \mathtt{mack} : \mathtt{\hat{}rw}[\mathtt{\hat{}w}[\mathtt{Site}\ \mathtt{\hat{}w}[]]]\ \textbf{in}\\
\quad @_D(\mathit{Daemon} \mid \mathsf{migReq}[a\ \mathtt{mack}] \mid \prod_i \mathsf{mesgReq}\{T_i\}\ [a\ c_i\ v_i])\\
\quad \mid @_a(\mathsf{migBlock}(\mathtt{mack}\ Q) \mid [\![P]\!]_a \mid \mathtt{Deliverer})
\end{array}$$

are temporary immobile under `lock` w.r.t. $((\Gamma, \Omega_D), M)$.

Lemma 6.4.4 (Blocked by local lock)
Given that Γ is a closed located type context, $\Gamma \vdash_a P$, $\vdash \Gamma, \Phi_{aux}$ and $M \subseteq \mathsf{agents}(\Gamma)/\{a\}$. $@_a(\llbracket P \rrbracket_a | \texttt{Deliverer})$ is temporarily immobile under `currentloc` w.r.t. $((\Gamma, \Phi_{aux}), M)$.

Lemma 6.4.5 (Blocked by acknowledgement)
Let Φ^-_{aux} be defined as follows.

$$\Phi^-_{aux} \stackrel{\text{def}}{=} \Phi_{aux}/\texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}$$

Given that Γ is a closed located type context, $\{\texttt{ack}, \texttt{currentloc}\} \cap \mathsf{fv}(P, Q) = \emptyset$, and $\Gamma,\ \Phi^-_{aux},\ \texttt{ack} : \texttt{\^{}rw}T \vdash LP$, where LP is defined below.

$$LP = \textbf{new}\ \texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}\ \textbf{in}\ @_a(\llbracket P \rrbracket_a \mid (\texttt{ack}\textbf{?}p \to Q) \mid \texttt{Deliverer})$$

LP is temporarily immobile under `ack` w.r.t. $((\Gamma,\ \Phi^-_{aux},\ \texttt{ack} : \texttt{\^{}rw}T), M)$.

The proofs of these lemmas are non-trivial, since the $\mathcal{C}$-encoding introduces many additional τ-steps, which can make it hard to determine all the possible derivatives. Many of these τ-steps, however, are confluent, which means that some of the derivatives are related by expansion — a useful property that cannot be exploited by Definitions 6.4.2-6.4.3. To deal with the additional confluent τ-steps, we use a technique similar to "up to" equivalences, allowing blocking sets to be closed up under translocating weak bisimulation.

Definition 6.4.4 (Blocking Set up to $\dot{\approx}$)
A translocating unary relation $\mathcal{M}$ is a *blocking set up to* $\dot{\approx}$ under lock l if, for any $LQ \in \mathcal{M}^M_\Gamma$, $M \subseteq \mathsf{mov}(\Gamma)$, $\Gamma \vdash LQ$, and valid relocator δ for (Γ, M), we have $\Gamma\delta \Vdash LQ \xrightarrow[\Theta]{\beta} LQ'$ implying the following:

- $\beta = \tau$ implies there exists LQ'' such that $LQ' \dot{\approx}^M_{\Gamma\delta} LQ''$ and $LQ'' \in \mathcal{M}^M_{\Gamma\delta}$;
- $\beta = @_b c\textbf{?}v$ with $l \notin \mathsf{fv}(\beta)$ implies there exists LQ'' such that $LQ' \dot{\approx}^{M \cup \mathsf{mov}(\Theta)}_{\Gamma\delta,\Theta} LQ''$ and $LQ'' \in \mathcal{M}^{M \cup \mathsf{mov}(\Theta)}_{\Gamma\delta,\Theta}$;
- $\beta = @_b c\textbf{!}v$ implies $l \notin \mathsf{fv}(\beta)$, and there exists LQ'' such that $LQ' \dot{\approx}^M_{\Gamma\delta,\Theta} LQ''$ and $LQ'' \in \mathcal{M}^M_{\Gamma\delta,\Theta}$; and
- $\beta \neq @_b\mathsf{migrate\ to}\ s$, for any b, s.

We have proved that if a process belongs to a blocking set up to $\dot{\approx}$ under l then it is temporarily immobile under the same lock. The details of the proof are in Appendix C on page 227.

Lemma 6.4.6 (Blocking set up to $\dot{\approx}$ implies temporary immobility)
Given that Γ is a closed located type context, and $M \subseteq \mathsf{agents}(\Gamma)$ is a translocating index, if there exists a blocking set up to $\dot{\approx}$ under l, $\mathcal{M}$, such that $LP \in \mathcal{M}_{\Gamma}^{M}$ then LP is temporary immobile under l w.r.t. (Γ, M).

By using blocking sets up to $\dot{\approx}$, we drastically cut down the size of the transition analysis required for proving Lemmas 6.4.4 to 6.4.3. The details of the proofs of properties related to $\mathcal{C}$-encoding can be found in Appendix C.1 on page 232. Other useful properties are that temporary immobility is preserved by parallel composition and **new**-binding, and by weak translocating bisimulation; the formal statements are given below. The details of the proofs of these properties can be found in Appendix C on page 230.

Lemma 6.4.7 (Composition preserve temporary immobility)
If LP and LQ are temporarily immobile under l, w.r.t. $((\Gamma, \Delta), M)$ with $l \in \mathsf{dom}(\Gamma)$ and Δ extensible, then **new** Δ **in** $(LP|LQ)$ is temporarily immobile under l, w.r.t. $(\Gamma, M \cap \mathsf{dom}(\Gamma))$.

Lemma 6.4.8 (Weak bisimulation preserves temporary immobility)
If LP is temporary immobile under l w.r.t. (Γ, M) and $LP \dot{\approx}_{\Gamma}^{M} LQ$ then LQ is temporary immobile under l w.r.t. (Γ, M).

6.5 Maps and Their Operators

The expressiveness of the π-calculus allows a wide-range of data structures to be encoded into its core syntax. Such data structures include lists [Mil93b], objects [Wal95], and B-trees [Phi96]. This section gives an encoding of finite maps, used in the example infrastructure for keeping track of current sites of registered agents. The encoding builds on that given in [SWP99], adding types. Maps are represented as linked data cells. Each cell is a replicated input on a channel — say m — and is either empty (represented by $\mathsf{emptyCell}(m)$) or containing an entry v_1 with a corresponding value v_2 and a pointer to the next cell m' (represented by $\mathsf{mapCell}(m, m', v_1, v_2)$). The encoding of map cells is given below.

$$\begin{array}{rcl} \mathsf{emptyCell}(m) & \stackrel{\text{def}}{=} & *m?[x\ \texttt{fnd}\ \texttt{nfd}] \rightarrow \texttt{nfd!}[] \\ \mathsf{mapCell}(m, m', v_1, v_2) & \stackrel{\text{def}}{=} & *m?[x\ \texttt{fnd}\ \texttt{nfd}] \rightarrow \\ & & \qquad \textbf{if } x = v_1 \textbf{ then } \texttt{fnd!}v_2 \textbf{ else } m'![x\ \texttt{fnd}\ \texttt{nfd}] \end{array}$$

A map cell m can be queried by sending a tuple $[v\ \texttt{fnd}\ \texttt{nfd}]$ along the channel m. If m is an empty cell, the "not found" channel `nfd` is signalled. If the cell is $\mathsf{mapCell}(m, m', v, v_2)$, the corresponding value v_2 is sent along the "found" channel `fnd`. Otherwise, if the cell is

$\mathsf{mapCell}(m, m', v_1, v_2)$ with $v_1 \neq v$, the tuple $[v\ \mathtt{fnd}\ \mathtt{nfd}]$ is forwarded to the next cell m', in effect searching the reminder of the linked cells.

Each map type $\mathtt{Map}[T_1\ T_2]$ is parameterised by two types: the type of its entries T_1 and the type of its entry values T_2. We translate map types to the core type syntax below.

$$\mathtt{Map}[T_1\ T_2] \stackrel{\text{def}}{=} \mathtt{\char`^rw}[T_1\ \mathtt{\char`^w}T_2\ \mathtt{\char`^w}[]]$$

Using these map cells, the encoding of the operators on maps is as follow.

$$\begin{array}{lcl}
c\mathtt{!}\,\mathbf{emptymap}[T_1\ T_2] & \stackrel{\text{def}}{=} & \mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (c\mathtt{!}m \mid \mathsf{emptyCell}(m)) \\
\mathbf{lookup}[T_1\ T_2]\ v\ \mathbf{in}\ m\ \mathbf{with} & \stackrel{\text{def}}{=} & \mathbf{new}\ \mathtt{fnd} : \mathtt{\char`^rw}T_2,\ \mathtt{nfd} : \mathtt{\char`^rw}[]\ \mathbf{in}\ (m\mathtt{!}[v\ \mathtt{fnd}\ \mathtt{nfd}] \\
\quad \mathbf{found}(p) \rightarrow P & & \mid \mathtt{fnd?}p \rightarrow P \\
\quad \mathbf{notfound} \rightarrow Q & & \mid \mathtt{nfd?}[] \rightarrow Q) \\
\multicolumn{3}{l}{\mathbf{let}[T_1\ T_2]\ m' = (m\ \mathbf{with}\ v_1 \mapsto v_2)\ \mathbf{in}\ P} \\
\multicolumn{3}{l}{\quad \stackrel{\text{def}}{=}\ \mathbf{new}\ m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ P \mid \mathsf{mapCell}(m, m', v_1, v_2)}
\end{array}$$

An output of an empty map via channel c is translated to an empty cell m in parallel with an output of a reference to (**new**-bound) m on c. Looking up an entry, say v, in the map simply exploits the map cell construct: two fresh channels $\mathtt{fnd}, \mathtt{nfd}$ are created and sent, together with v, to the head of the linked cells m. Such a message transverses the linked cells until it reacts with a cell whose entry matches the queried entry v; in this case, the value, say v', corresponding to v will be sent along `fnd`. This output on `fnd` triggers the execution of P with the appropriate parts of the value v' bound to the parameters in the pattern p. If no such cell exists, however, the channel `nfd` will be signalled, in effect triggering the execution of Q. Updating a map, whether adding a new entry or updating an existing entry, causes a new map cell to be appended in front of the linked cells. This is clearly not efficient in terms of storage for cells of out-of-date entries are never garbage-collected.

In the rest of this section, we shall prove that this encoding of finite maps is correct. This involves thinking of the linked cells which implement a map as a list of pairs (possibly duplicated). We show how a map can be constructed from a list of pairs, and then show how map operations affect (or use) such a list.

Firstly, we need to check the elements of ls with respect to some type context, to ensure that the process constructed from a list ls is well-formed. We then define a function that turn a list of pairs into the corresponding Nomadic π process.

Definition 6.5.1 (Well-typed list)
A list ls consists of pairs of types T_1 and T_2 w.r.t. Γ, written $\Gamma \vdash ls \in \mathtt{List}\ T$, if for all element e in ls, $\Gamma \vdash e \in T$.

Definition 6.5.2 (Making Maps)
Given a closed located type context Γ, the function $\mathsf{makeMap}(m; ls)$ takes a list of pairs ls, such that $\Gamma \vdash ls \in \mathtt{List}\ [T_1\ T_2]$, to a basic process P, accessible via channel m of type $\mathtt{Map}[T_1\ T_2]$. It is defined recursively as follows:

$$\begin{array}{rcl} \mathsf{makeMap}(m; \mathbf{nil}) & \stackrel{\text{def}}{=} & \mathsf{emptyCell}(m) \\ \mathsf{makeMap}(m; [v_1\ v_2]\mathbf{::}ls) & \stackrel{\text{def}}{=} & \mathbf{new}\ m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ \mathsf{makeMap}(m'; ls) \\ & & \quad \mid \mathsf{mapCell}(m, m', v_1, v_2) \end{array}$$

We prove that our definition of map cell and linking is well-formed.

Lemma 6.5.1 (Map construction preserves typing)
For a closed located type context Γ, if $\Gamma \vdash ls \in \mathtt{List}\ [T_1\ T_2]$ and $m \notin \mathsf{dom}(\Gamma)$ then $\Gamma,\ m : \mathtt{Map}[T_1\ T_2] \vdash \mathsf{makeMap}(m; ls)$.

Proof: An induction on the size of ls. ■

It is not hard to prove that the result of a map update is structurally congruent to a map represented by the appended list.

Lemma 6.5.2 (Map update induces structural congruence)
If $m' \notin \mathsf{fv}(ls)$ and $m \notin \mathsf{fv}(P)$ then

$$\begin{array}{l} @_a\mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m; ls) \mid \mathbf{let}\ m' = (m\ \mathbf{with}\ v \mapsto v')\ \mathbf{in}\ P) \\ \qquad \equiv @_a\mathbf{new}\ m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m'; [v\ v']\mathbf{::}ls) \mid P). \end{array}$$

Proof: All we need do is to expand definitions and apply structural congruence.

$$\begin{array}{rcll} LHS & = & @_a\mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m; ls) & \\ & & \quad \mid \mathbf{new}\ m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{mapCell}(m', m, v, v') \mid P)) & \text{since } m' \notin \mathsf{fv}(ls) \\ & \equiv & @_a\mathbf{new}\ m, m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m; ls) \mid \mathsf{mapCell}(m', m, v, v') \mid P) & \\ & \equiv & @_a\mathbf{new}\ m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (P & \\ & & \quad \mid \mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m; ls) \mid \mathsf{mapCell}(m', m, v, v'))) & \\ & & & \text{since } m \notin \mathsf{fv}(P) \\ & = & @_a\mathbf{new}\ m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (P \mid \mathsf{makeMap}(m'; [v\ v']\mathbf{::}ls)) & \end{array}$$

■

The correctness of the map encoding relies on the fact that when looking up an entry, the linked list is being searched from its head to its tail. The most recent entry will therefore be found first. We define a function $\mathsf{lookupL}(v;ls)$, which searches for the *first* entry $[v\ v'']$ in the list ls and returns v''. This is formally defined as follows.

$$\begin{array}{rcll}
\mathsf{lookupL}(v;\mathbf{nil}) & \stackrel{\text{def}}{=} & \text{undefined} & \\
\mathsf{lookupL}(v;[v\ v'']\mathbf{::}ls) & \stackrel{\text{def}}{=} & v'' & \\
\mathsf{lookupL}(v;[v'\ v'']\mathbf{::}ls) & \stackrel{\text{def}}{=} & \mathsf{lookupL}(v;ls) & v \neq v'
\end{array}$$

The lemma below shows the correctness of the map lookup operation.

Lemma 6.5.3 (Map lookup yields an expansion)
Given a closed located type context Γ and $\Gamma \vdash ls \in \texttt{List}\ [T_1\ T_2]$, if m is output-only in P and in Q w.r.t. $\Gamma, m : \texttt{Map}[T_1\ T_2]$ and

$$\begin{array}{ll}
\Gamma \vdash @_a\mathbf{new}\ m : \texttt{Map}[T_1\ T_2]\ \mathbf{in}\ \mathsf{makeMap}(m;ls) \mid & \mathbf{lookup}[T_1\ T_2]\ v\ \mathbf{in}\ m\ \mathbf{with} \\
& \quad \mathbf{found}(p) \to P \\
& \quad \mathbf{notfound} \to Q
\end{array}$$

then the following hold.

- $v \in \mathsf{dom}(ls)$ and $\mathsf{lookupL}(v;ls) = v'$ implies

$$\begin{array}{ll}
@_a\mathbf{new}\ m : \texttt{Map}[T_1\ T_2]\ \mathbf{in}\ \mathsf{makeMap}(m;ls) \mid & \mathbf{lookup}[T_1\ T_2]\ v\ \mathbf{in}\ m\ \mathbf{with} \\
& \quad \mathbf{found}(p) \to P \\
& \quad \mathbf{notfound} \to Q \\
\multicolumn{2}{l}{\quad \succeq_\Gamma\ @_a\mathbf{new}\ m : \texttt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m;ls) \mid \mathsf{match}(p, v')P).}
\end{array}$$

- $v \notin \mathsf{dom}(ls)$ implies

$$\begin{array}{ll}
@_a\mathbf{new}\ m : \texttt{Map}[T_1\ T_2]\ \mathbf{in}\ \mathsf{makeMap}(m;ls) \mid & \mathbf{lookup}[T_1\ T_2]\ v\ \mathbf{in}\ m\ \mathbf{with} \\
& \quad \mathbf{found}(p) \to P \\
& \quad \mathbf{notfound} \to Q \\
\multicolumn{2}{l}{\quad \succeq_\Gamma\ @_a\mathbf{new}\ m : \texttt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m;ls) \mid Q).}
\end{array}$$

Proof: The proof of this lemma easily follows from this result: given that $\Gamma \vdash ls \in \texttt{List}\ [T_1\ T_2]$ and that a process m is output-only in R w.r.t. $\Gamma, m : \texttt{Map}[T_1\ T_2]$, we have:

- if $v \in \mathsf{dom}(ls)$ then

$$\begin{array}{l}
@_a\mathbf{new}\ m : \texttt{Map}[T_1\ T_2]\ \mathbf{in}\ (R|\mathsf{makeMap}(m;ls)|m\mathbf{!}[v\ f\ n]) \\
\qquad \succeq_\Gamma\ @_a\mathbf{new}\ m : \texttt{Map}[T_1\ T_2]\ \mathbf{in}\ (R|\mathsf{makeMap}(m;ls)|f\mathbf{!}\mathsf{lookupL}(v;ls))
\end{array}$$

- if $v \notin \mathsf{dom}(ls)$ then

$$\begin{aligned} &@_a\mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (R|\mathsf{makeMap}(m;ls)|m\mathbf{!}[v\ f\ n]) \\ &\qquad \succeq_\Gamma\ @_a\mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (R|\mathsf{makeMap}(m;ls)|n\mathbf{!}[]) \end{aligned}$$

whenever the LHS process is well-typed w.r.t. Γ and $M \subseteq \mathsf{mov}(\Gamma)$.

The proof of this uses an induction on the size of ls. We do not need to define an expansion relation here, since we can use the earlier proof techniques for deriving this result. In particular, channel m is functional, so the techniques of Section 6.2.2 can be applied. To demonstrate this, we include the inductive case when $v \in \mathsf{dom}(ls)$. Let $ls = [v_1\ v_2]\mathbf{::}ls'$ with $v_1 \neq v$ and $\mathsf{lookupL}(v;ls') = v'$. We have:

$$\mathsf{makeMap}(m;ls) \quad = \quad \mathbf{new}\ m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (\mathsf{makeMap}(m';ls') \mid \mathsf{mapCell}(m,m',v_1,v_2)).$$

Let LP denote $@_a\mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (R|\mathsf{makeMap}(m;ls)|m\mathbf{!}[v\ f\ n])$, we have:

$$\begin{array}{lll} LP\ \succeq_\Gamma & @_a\mathbf{new}\ m,m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (R & \text{By Lemma 6.2.3} \\ & \quad \mid \mathsf{makeMap}(m';ls') \mid \mathsf{mapCell}(m,m',v_1,v_2) & \\ & \quad \mid \mathbf{if}\ v = v_1\ \mathbf{then}\ f\mathbf{!}v_2\ \mathbf{else}\ m'\mathbf{!}[v\ f\ n]) & \\ \quad\ \succeq_\Gamma & @_a\mathbf{new}\ m,m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (R & \text{By Lemma 6.3.1} \\ & \quad \mid \mathsf{makeMap}(m';ls') \mid \mathsf{mapCell}(m,m',v_1,v_2) & \\ & \quad \mid m'\mathbf{!}[e\ f\ n]) & \\ \quad\ \succeq_\Gamma & @_a\mathbf{new}\ m,m' : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (R & \text{by the induction hypothesis} \\ & \quad \mid \mathsf{makeMap}(m';ls') \mid \mathsf{mapCell}(m,m',v_1,v_2) & \\ & \quad \mid f\mathbf{!}v') & \\ \quad\ = & @_a\mathbf{new}\ m : \mathtt{Map}[T_1\ T_2]\ \mathbf{in}\ (R \mid \mathsf{makeMap}(m;ls) \mid f\mathbf{!}v') & \end{array}$$

Clearly, $\mathsf{lookupL}(e;ls) = v'$, thus true for this case.

Other cases are omitted. ■

We may also prove that the $\mathsf{lookupL}(v;ls)$ indeed finds the first pair with the entry v in the list ls. The function $\mathsf{consolidate}(ls)$ defined below removes duplicate entries in the list ls, so that only the most recent update remains for each entry.

$$\begin{aligned} \mathsf{consolidate}(ls) &\stackrel{\text{def}}{=} \mathsf{deflate}_\emptyset(ls) \\ \mathsf{deflate}_S(\mathbf{nil}) &\stackrel{\text{def}}{=} \mathbf{nil} \\ \mathsf{deflate}_S([v\ v']\mathbf{::}ls) &\stackrel{\text{def}}{=} \begin{cases} [v\ v']\mathbf{::}\mathsf{deflate}_{S,v}(ls) & v \notin S \\ \mathsf{deflate}_S(ls) & v \in S \end{cases} \end{aligned}$$

We prove that $\mathsf{lookupL}(v; ls) = \mathsf{lookupL}(v; \mathsf{consolidate}(ls))$ for any v. This result confirms the correctness of the map encoding.

Chapter 7

The Correctness Proof

This chapter focuses on the correctness proof of the centralised server translation. The operational equivalences given in Chapter 5 can be used for directly comparing the behaviour of a source program and its encoding, since $\mathrm{n}\pi_{\mathsf{LD}}$ is a fragment of $\mathrm{n}\pi_{\mathsf{LD,LI}}$. In particular, we use a coupled simulation, since the $\mathcal{C}$-encoding introduces partial commitment (an example of which is given in Section 5.4.2).

It is infeasible, however, to directly write down relations which form a coupled simulation, since our example infrastructure introduces many τ steps — making exhaustive enumeration of derivatives difficult. Each of the additional τ step induces an *intermediate state*: a target level term which is not a literal translation of any source level term. Some of these steps are deterministic *house-keeping* steps, eg. looking up a site in the site map; they can be reduced to (and related by expansions to) *normal forms*. Terms are said to be in the normal form if they have no house-keeping steps. Some, however (migration steps and acquisitions of the daemon lock or of local agent locks), are *partial commitment steps*. They involve nondeterministic internal choices and lead to *partially committed states*: target level terms which are not bisimilar to any source level term, but must be related to them by coupled simulation.

We factor the construction of the main coupled simulation (between a source program and its encoding) by introducing an *intermediate language* IL. This helps us manage the complexity of the state-space of the encoding, by:

1. reducing the size of the relation, omitting states which reduce with house-keeping steps to certain normal forms; and

2. dealing with states in which many agents may be partially committed simultaneously; and

3. capturing some invariants, eg. that the daemon's site-map is correct, in a type system for IL.

The cost is that the typing and labelled transition rules for IL must be defined. The infrastructure encoding is factored into the composition of a *loading* encoding $\mathcal{L}$, mapping source terms to corresponding terms in the intermediate language, and a *flattening* encoding $\mathcal{F}$, mapping terms in the intermediate language to their corresponding target terms.

$$
\begin{tikzcd}
\mathrm{n}\pi_{\mathsf{LD,LI}} \arrow[r, "\mathcal{L}[\![\cdot]\!]"] \arrow[dr, "\mathcal{C}[\![\cdot]\!]"'] & \mathrm{IL} \arrow[d, "\mathcal{F}[\![\cdot]\!]"] \\
& \mathrm{n}\pi_{\mathsf{LD}}
\end{tikzcd}
\qquad
\mathrm{n}\pi_{\mathsf{LD,LI}} \xleftarrow[\mathcal{D}^\sharp[\![\cdot]\!]]{\mathcal{D}^\flat[\![\cdot]\!]} \mathrm{IL}
$$

We use two functions mapping intermediate language states back into the source language. The *undo* and *commit* decoding functions, $\mathcal{D}^\flat$ and $\mathcal{D}^\sharp$ respectively, undo and complete partially committed actions.

Using these encoding and decoding functions, we are able to construct behavioural relations among the source, the intermediate and the target language. Two key properties are:

1. $\mathcal{F}$ is an expansion between the intermediate language terms and target terms; and

2. $(\mathcal{D}^\flat, \mathcal{D}^\sharp)$ is a coupled simulation between the intermediate language terms and source terms.

In proving the former result, we essentially deal with all the house-keeping steps, relating terms introduced by such steps to some normal forms. Such normal forms allow house-keeping steps to be abstracted away, so that in proving the latter result, we can concentrate on relating partially-committed terms to target-level terms. The two key results can then be combined to provide a coupled simulation on source and target terms. The observation that every source term LP and its translation $\mathcal{C}[\![LP]\!]$ are related by this coupled simulation concludes the proof of correctness of $\mathcal{C}$-encoding.

This chapter is organised as follows. We start with Section 7.1, giving an overview of correctness proofs of encodings in process calculi literature. In Section 7.2, we describe the intermediate language, giving its syntax, type system and labelled transition semantics. In Section 7.3 and Section 7.4, we give the loading and flattening encodings as well as the two decoding functions, for relating the source, intermediate language and target terms. Behavioural properties of such functions are shown using the proof techniques developed in

Chapters 5 and 6. Section 7.5 gives the correctness statement, directly relating the source and target terms. The proof of this statement combines the behavioural properties proved for the encoding and decoding functions.

7.1 Background

In theoretical computer science, to study a particular problem it is often more convenient to introduce a variant of existing calculi, or to design a specific calculus, instead of relying on established formalisms. Differences in design choices and styles — syntax, type system and operational semantics — of these particular calculi are difficult to compare. A formal way to assess the similarities and differences between two calculi (typically a novel calculus and an established calculus) is via encodings, accompanied by proofs of some notion of correctness. Here, in this thesis, we regard an *encoding* or a *translation*, ranged over by $[\![\cdot]\!]$, as a function from terms in a *source* calculus to terms in a *target* calculus.

There exists a large body of literature on encodings. Here we shall focus on the encodings which are related to the π-calculus and its variants, some examples of which include: various encodings of λ-calculi into π-calculi [Mil92, San93b, San94b], the encoding of concurrent object-based language POOL in π-calculus [Wal95], the encoding of choice into a choice-free π-calculus [Nes96], the encodings of the π calculus to and from the Join calculus [FG96], and the encodings of various calculi to the fusion calculus [Vic98]. In this brief overview, we shall discuss a few themes common to the encodings in the context of process calculi. We refer to the thesis of Nestmann [Nes96] and that of Fournet [Fou98] for further details and discussion.

Direct Correspondence vs. Full Abstraction Intuitively, for an encoding $[\![]\!]$ to be correct, we require that every source term P (perhaps being well-formed in some way) and its translation $[\![P]\!]$ should exhibit the same behaviour. There are two common styles of formally stating the correctness of encodings. The first style directly relates the source and the translated terms by some operational equivalence — a style which we shall refer to as *direct correspondence*. To state this type of result, the source and the target calculi must have the same form of operational semantics. The encodings with direct correspondence correctness result are typically *internal*, ie. with the target calculus a fragment of the source calculus. The examples of such encodings include the choice encodings of [NP96], translating the π-calculus with input-only choice into its choice-free fragment, and the translation of the choice-free asynchronous π-calculus into trios [Par99]. Since the example infrastructure is internal (the low-level Nomadic π-calculus is a fragment of the high-level), the correctness result given in this chapter is also of this style.

In general, however, the semantics of the source and the target calculi may differ, so that the behaviour of their terms cannot be directly related. The second alternative is to express the correctness of an encoding as the preservation and reflection of equivalence of source terms. This is known as *full abstraction*; a typical statement is as below:

$$P \simeq_s Q \quad \text{if and only if} \quad [\![P]\!] \simeq_t [\![Q]\!]$$

where $\simeq_s$ and $\simeq_t$ denote equivalences of the source and the target calculi respectively. If the above statement can be proved, we may say that the source calculus is *at least as expressive* as the target calculus. There are numerous examples of encodings whose correctness statements require full abstraction. Some examples of these are:

- various encodings of the λ-calculus to the π-calculus [Mil92, San94a, San95a],
- an encoding of higher-order π-calculus to π-calculus [San93b],
- the encoding of the choice-free asynchronous π-calculus into the join-calculus [FG96],
- the encoding of a variant of the join calculus with authentication primitives to another variant with cryptographic primitives [AFG00].

Full abstraction results may be less convincing than direct correspondence results, but are necessary eg. when the translation is from the λ-calculus to the π-calculus, or when the notions of barbs of the two calculi are defined differently.

There exists many plausible notions of operational equivalences and preorders that one may choose for stating the correctness result (whether it is in the direct correspondence form, or in the full abstraction). Such notions include strong and weak bisimulation, expansion, coupled simulation, testing and trace equivalence. The choice of an equivalence is generally closely related to the particular encodings. However, for the correctness result to be convincing, the operational relations relating the source and the translated terms should be as strong as possible. Furthermore, for the proof to be tractable, such an equivalence should have convenient proof techniques. For these reason, simulation-based equivalences such as expansion, bisimulation and coupled simulation are generally employed for proving encodings correct.

Operational Correspondence To prove a correctness statement (whether it is in the direct correspondence or full abstraction form), we are often require some *operational cor-*

respondence results, in essence demonstrating a simulation-like relation between the source and the target terms. For example, we may show that an encoding $[\![\cdot]\!]$ preserves a reduction up to some equivalence $\simeq_t$, formally:

$$P \rightarrow_s Q \text{ implies } [\![P]\!] \Rightarrow_t \simeq_t [\![Q]\!]$$

where $\rightarrow_s$ and $\rightarrow_t$ denote reduction relation of the source and the target calculi respectively, and $\Rightarrow_t = \rightarrow_t^*$. The reduction relations of the above statement can be replaced by labelled transition relations if the source and the target calculi share the same form of LTS (typically when the encoding is internal). This is referred to as a *completeness* result in [Nes96]. The converse of the above result is referred to as a *soundness* result, which reflects the steps committed by the target-level terms back to those of source-level terms. As discussed in [Nes96], there are many plausible ways of stating a soundness result, an example of which is below:

$$[\![P]\!] \Rightarrow_t \simeq_t Q_t \text{ implies } \exists Q \,.\, P \rightarrow_s Q \;\wedge\; Q_t \simeq_t [\![Q]\!]$$

In some cases, where one of the calculi is not equipped with a notion of operational equivalence, eg. in [PV98], the operational correspondence results (both completeness and soundness) may be considered sufficient.

Uniformity According to Palamidessi [Pal97], an encoding $[\![\cdot]\!]$ is *uniform* if it satisfies the following for any processes P, Q and substitution σ:

$$[\![P|Q]\!] \;=\; [\![P]\!] \,|\, [\![Q]\!] \qquad [\![\sigma(P)]\!] \;=\; \sigma([\![P]\!])$$

The uniformity property often simplifies proofs of correctness, especially if the operational equivalences employed are also congruences, since it facilitates use of induction on the syntax of terms. Encodings which benefit from this are, eg. the choice encodings of Nestmann. Our example infrastructure is not uniform, though, as it introduces a centralised daemon at top level. This means that our reasoning must largely be about the whole system, dealing with interactions between encoded agents and the daemon. We cannot use simple induction on source program syntax.

7.2 The Intermediate Language

This section introduces the intermediate language IL, which provides an abstraction of target-level reductions. We first give the syntax and its informal description; the type system and the formal semantics of the intermediate language are given in Section 7.2.1 and Section 7.2.2 respectively. A term in the intermediate language is referred to as a *system*, describing the states of the encoded agents and of the daemon. Each system represents a normal form of target-level derivatives, possibly in a partially committed state. The syntax is:

$$Sys \quad ::= \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$$

Each system $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ is parameterised by Δ, a located type context corresponding to all names dynamically created during the execution of the program, and $\boldsymbol{D}$ and $\boldsymbol{A}$, the state of the daemon and of the agents. Δ is binding in $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$, and is therefore subject to alpha-conversion. The latter two parameters are described in more detail below.

- The state $\boldsymbol{D}$ of the daemon is a term of the following grammar:

$$\begin{array}{lll} \boldsymbol{D} & ::= & [map\ \mathsf{mesgQ}] \\ \mathsf{mesgQ} & ::= & \prod_{i \in I} \mathsf{mesgReq}(\{T_i\}\,[a_i\ c_i\ v_i]) \qquad I \text{ is an index set} \end{array}$$

  Each daemon state $[map\ \mathsf{mesgQ}]$ consists of two components: a site map, map, expressed as a list of pairs; and a *message queue* mesgQ, expressed as a parallel composition of message forwarding requests, indexed by I. A message forwarding request $\mathsf{mesgReq}(\{T\}\,[a\ c\ v])$ requires the daemon to forward $c!v$ to the agent a, where T is the type of v. The definition of $\mathsf{mesgReq}$ as a process is given in Figure 2.3.

- The state $\boldsymbol{A}$ of the agents is a partial function mapping agent names to agent states. Each agent state, represented as $[P\ \boldsymbol{E}]$, consists of a *main body* P and a *pending state* $\boldsymbol{E}$. The syntax of $\boldsymbol{E}$ is given below:

$$\boldsymbol{E} \quad ::= \quad \mathsf{FreeA}(s) \ \mid\ \mathsf{RegA}(b\ Z\ s\ P\ Q) \ \mid\ \mathsf{MtingA}(s\ P) \ \mid\ \mathsf{MrdyA}(s\ P)$$

  The pending states basically describe the state of the local lock in an agent. If an agent a has pending state $\mathsf{FreeA}(s)$, the local lock of a is free. Otherwise, the local lock of a is acquired by $\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q$ (when its state is $\mathsf{RegA}(b\ Z\ s\ P\ Q)$) or $\mathbf{migrate\ to}\ s \to P$ (when its state is $\mathsf{MtingA}(s\ P)$ or $\mathsf{MrdyA}(s\ P)$). In $\mathsf{FreeA}(s)$ and $\mathsf{RegA}(b\ Z\ s\ P\ Q)$, s denotes the current site of a, internally recorded and maintained by the agent itself. In $\mathsf{RegA}(b\ Z\ s\ P\ Q)$, the name b is bound in P and Q and is subject to alpha-conversion.

Informally, the transitions of a system can be classified into three classes: local computation, initialising request, and request processing. The description of each kind of transition is given below.

Local computation A process from the main body of an agent may be executed immediately if it is an **iflocal**, **if**, **let** or a pair of an output and a (replicated) input on the same channel. The result of such an execution (as described by the LTS of Nomadic-π) is placed in parallel with other processes in the main body. These steps correspond exactly to those taken by source- and target-level terms.

Initialising Request This deals with execution which produces a registration, migration, or message forwarding request. A process $\texttt{create}^Z\ b = P\ \texttt{in}\ Q$ or $\texttt{migrate to}\ s \rightarrow P$ from the main body of a may begin execution if the local lock is free, ie. the pending state is $\mathsf{FreeA}(s')$. The result of such initiation turns the pending state to $\mathsf{RegA}(b\ Z\ s'\ P\ Q)$ or $\mathsf{MtingA}(s\ P)$ respectively. Translating into target-level terms, an agent in such a state has successfully acquired its local lock and sent a registration or migrating request to the daemon. To be more precise, we may use the definitions of regReq, migReq, and regBlockC, regBlockP, migBlock, given in Figures 2.3 and 2.4. In the creation case, the daemon reacts to an output on `register`, spawning $\mathsf{regReq}(b\ s\ \texttt{ack})$; agents b (newly created) and a become $\mathsf{regBlockC}(s\ \texttt{pack}\ \texttt{rack}\ P)$ and $\mathsf{regBlockP}(s\ \texttt{pack}\ Q)$, waiting for an acknowledgement from the daemon and from b, respectively. In the migration case, the daemon reacts to an output on `migrating`, spawning $\mathsf{migReq}(a\ \texttt{ack})$; a becomes $\mathsf{migBlock}(s\ \texttt{mack}\ P)$, waiting for the permission to migrate from the daemon.

Executing an LI $\langle b@?\rangle c!v$ from the main body of a may begin execution regardless of the pending state of a. The execution results in the message forwarding request $\mathsf{mesgReq}(\{T\}\,[b\,c\,v])$ being added to the message queue of the daemon (T is the type of v in the context). In the target-level terms, this corresponds to the daemon reacting to an output on `message`, spawning $\mathsf{mesgReq}(\{T\}\,[b\ c\ v])$.

Request Processing This deals with the message forwarding requests in the daemon, and the pending states of agents (when they are not $\mathsf{FreeA}(s)$).

- A system with a message forwarding request $\mathsf{mesgReq}(\{T\}\,[b\ c\ v])$ executes a single reduction step, corresponding in the target-level to acquiring the daemon lock, looking up the site of b, delivering the message, and receiving an acknowledgement from b. After completion, the message $c!v$ is added to the main body of b.
- A system with an agent in the registration request state $\mathsf{RegA}(b\ Z\ s\ P\ Q)$ has a single reduction step, corresponding, in the target-level, to acquiring the daemon lock, updating the site map and sending the acknowledgement to b. After completion, the

declaration $b : \texttt{Agent}^Z@s$ is placed at the top level and, at the same time, the site map is updated with the new entry $[b\ s]$. The new agent b, with the state $[P\ \mathsf{FreeA}(s)]$, now commences its execution, and so does its parent.

- Serving a migrating request $\mathsf{MtingA}(s\ P)$ from an agent a, however, involves two steps. The first step acquires the daemon lock, initialising the request, and turns the pending state of a to $\mathsf{MrdyA}(s\ P)$. In the second step, the agent a migrates to s (hence changes the top-level declaration) and the site map is updated with the entry $[a\ s]$. The first step corresponds in the target-level to acquiring the daemon lock, looking up the site of a in the site map, and sending an acknowledgement, thus permitting a to migrate. The daemon becomes $\mathsf{migProc}(a\ m\ \texttt{migrated})$, waiting for an acknowledgement from the agent a; a becomes $\mathsf{migReady}(s\ \texttt{migrated}\ P)$, and is now ready to migrate. The second step corresponds in the target-level to a migrating to s and sending an acknowledgement back to the daemon, which updates its site map and then sends the final acknowledgement to a, allowing it to proceed.

Figure 7.1 relates the states of intermediate terms to the infrastructure algorithm. It shows how the pending states correspond to the inter-agent communications involved in delivering an LI output, the migration, and the creation of a single agent, repeating the diagrams given in Section 2.3. Figure 7.2 gives the correspondence between steps in the source, intermediate and the target languages in the creation, migration and location-independent messaging cases. In the figure, some τ communication steps are annotated with the command or the name of the channel involved.

Note that when an agent, say a, is in the state $\mathsf{MrdyA}(s\ P)$, the daemon has already acquired the daemon lock, and is waiting for an acknowledgement from a. This has the following consequences.

- Since the $\mathcal{C}$-encoding preserves the invariant that at any time there is at most one output on the daemon lock, this means that, apart from a, there can be no other agent in the state $\mathsf{MrdyA}(s'\ P')$, for any s', P'.
- To proceed with any other request, the daemon lock must first be acquired. This means that while a is in such a state, no other request may proceed.

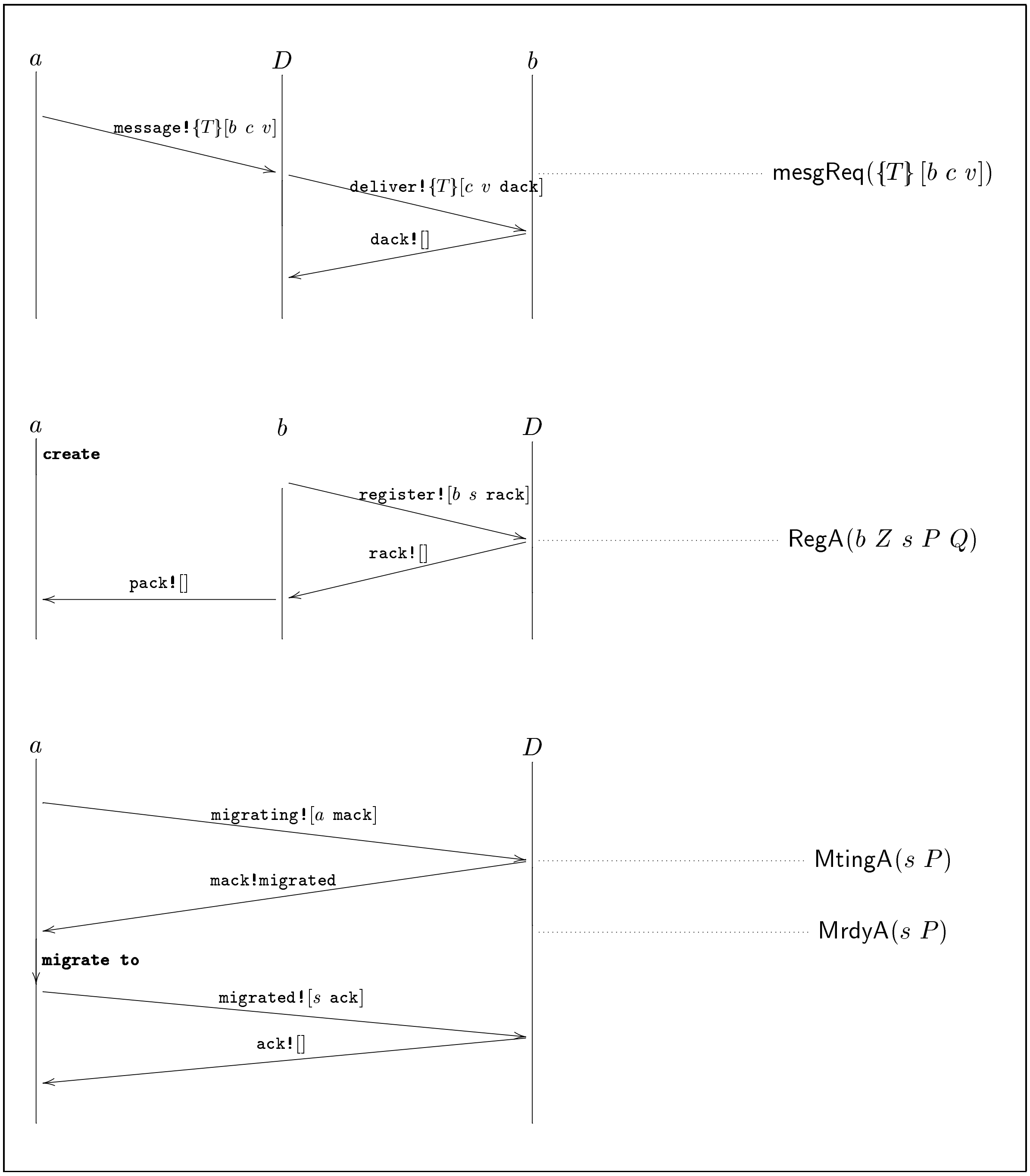


Figure 7.1: Relationship between the pending states and interaction between agents and the daemon

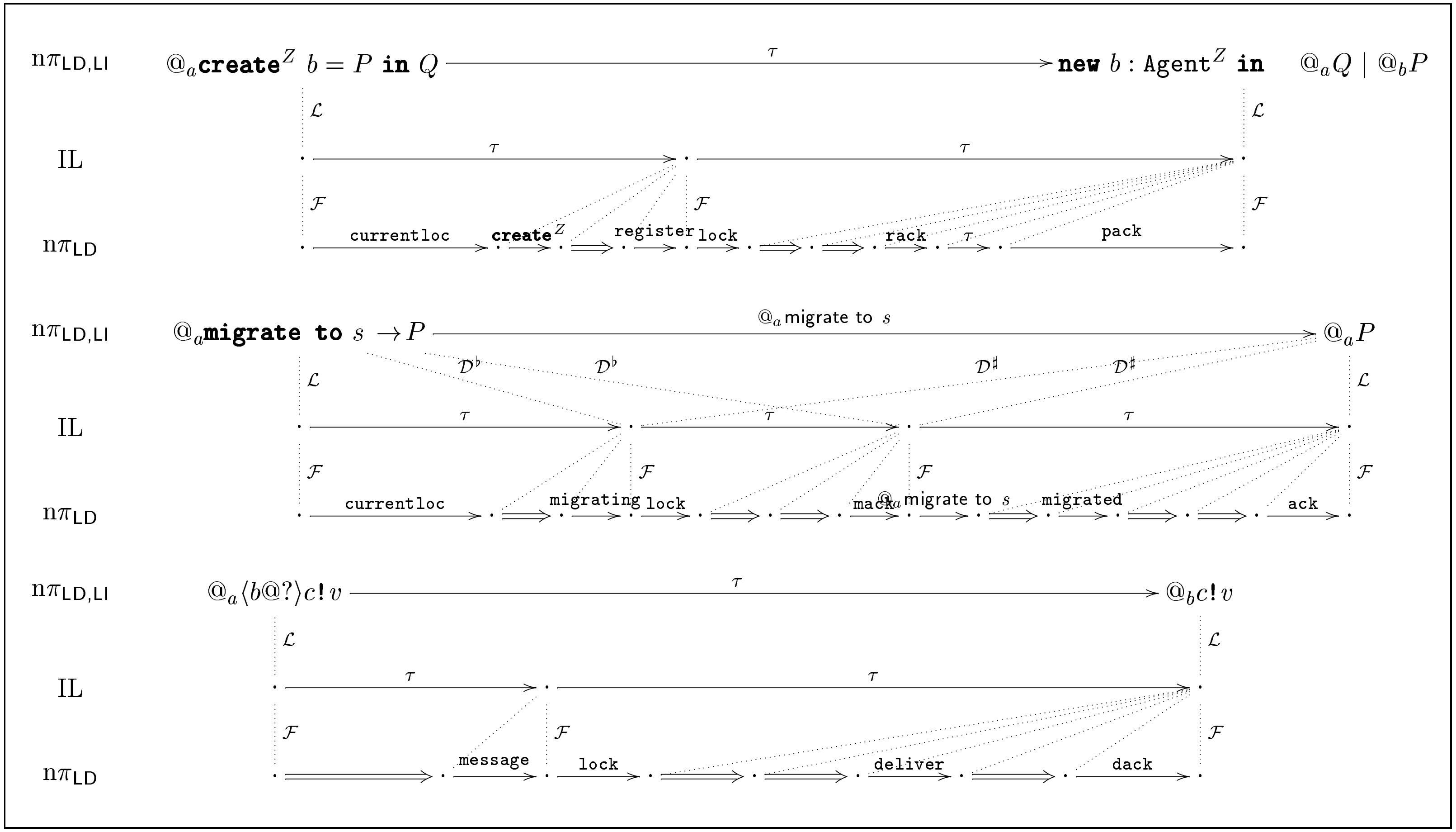


Figure 7.2: Relationship between source, intermediate, and target languages

The steps involved in serving registration and message forwarding requests are all house-keeping steps, inducing expansions. We therefore do not need to represent terms in which the daemon is busy serving such requests, for they can be related by expansion to terms in which such requests have been fully committed. On the other hand, the migration step that an agent in the state $\mathsf{MrdyA}(s\ P)$ may perform is not a house-keeping step (and does not induce expansion). It is therefore necessary to represent the state in which the daemon is busy serving a migration request. We refer to a system whose daemon lock is not available as a *busy* system, formally defined below.

Definition 7.2.1 (Busy and Idle Systems)
A system $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ is said to be *busy* if there exists $a \in \mathsf{dom}(\boldsymbol{A})$ such that $\boldsymbol{A}(a) = \mathsf{MrdyA}(s\ P)$ for some s, P. It is said to be *idle* otherwise.

In the process calculi literature, intermediate languages are often used for structuring proofs of encoding correctness. This includes the intermediate level of the Higher-Order π-calculus [San94a] for an encoding of the λ-calculus into the π-calculus, and the π_v calculus — a calculus with primitive values, higher-order abstractions and first-order interaction [LW95] — for an encoding of object-based languages into the π-calculus. Our work has been informed by the annotated choice language, introduced in Nestmann's work on choice encodings [Nes97, NP96]. As here, the intermediate language is used for dealing with partial commitment, although the uniformity of the encodings allows the language to be considerably simpler.

7.2.1 Type System

We formulate a type system for the intermediate language for ensuring the typability of processes contained in the main body of each agent in a system, as well as for capturing some invariants, eg. that the daemon's site map is correct. The typing rules of the intermediate language allows typing judgements of the following forms to be derived:

$\vdash \Phi\ ok$	located type context Φ is a valid system context
$\Phi \vdash map\ ok$	site map map is well-formed w.r.t. Φ
$\Phi \vdash \mathsf{mesgQ}\ ok$	message queue mesgQ is well-formed w.r.t. Φ
$\Phi \vdash \boldsymbol{A}\ ok$	state $\boldsymbol{A}$ of agents is well-formed w.r.t. Φ
$\Phi \vdash \boldsymbol{E}\ ok$	pending state $\boldsymbol{E}$ is well-formed w.r.t. Φ
$\Phi \vdash Sys\ ok$	system Sys is well-formed w.r.t. Φ

Note that we prove the coupled simulation over programs which are well-typed with respect to a *valid system context*, formally defined below.

(Sys-T-Map)

$$\frac{\Theta \vdash \mathit{map} \in \texttt{List}\ [\texttt{Agent}^{\texttt{s}}\ \texttt{Site}] \ \wedge\ \mathsf{consolidate}(\mathit{map}) = [a_1\ s_1]\texttt{::}\ldots\texttt{::}[a_n\ s_n]\texttt{::}\mathbf{nil} \wedge \\ \{a_1,\ldots,a_n\} = \mathsf{agents}(\Theta)\ \wedge\ \forall i \in \{1,\ldots,n\}\ .\ \Theta \vdash a_i @ s_i}{\Theta \vdash \mathit{map}\ \mathit{ok}}$$

(Sys-T-MesgQ)

$$\frac{\mathsf{mesgQ} = \prod_{i\in I} \mathsf{mesgReq}(\{T_i\}\,[a_i\ c_i\ v_i]) \quad \forall i \in I\ .\ \Theta \vdash [a_i\ c_i\ v_i] \in [\texttt{Agent}^{\texttt{s}}\ \texttt{\^{}w}T_i\ T_i]}{\Theta \vdash \mathsf{mesgQ}\ \mathit{ok}}$$

(Sys-T-FreeA)

$$\frac{\Theta \vdash a@s}{\Theta \vdash_a \mathsf{FreeA}(s)\ \mathit{ok}}$$

(Sys-T-RegA)

$$\frac{\Theta,\ b : \texttt{Agent}^Z@s \vdash_b P \quad \Theta,\ b : \texttt{Agent}^Z@s \vdash_a Q \\ \Theta \vdash a@s \quad \vdash \Theta,\ b : \texttt{Agent}^Z@s}{\Theta \vdash_a \mathsf{RegA}(b\ Z\ s\ P\ Q)\ \mathit{ok}}$$

(Sys-T-MigratingA)

$$\frac{\Theta \vdash_a P \quad \Theta \vdash s \in \texttt{Site}}{\Theta \vdash_a \mathsf{MtingA}(s\ P)\ \mathit{ok}}$$

(Sys-T-MigratedA)

$$\frac{\Theta \vdash_a P \quad \Theta \vdash s \in \texttt{Site}}{\Theta \vdash_a \mathsf{MrdyA}(s\ P)\ \mathit{ok}}$$

(Sys-T-AState)

$$\frac{\forall a \in \mathsf{dom}(\boldsymbol{A})\ \exists P, \boldsymbol{E}\ .\ \boldsymbol{A}(a) = [P\ \boldsymbol{E}]\ \wedge\ \Theta \vdash_a P\ \wedge\ \Theta \vdash_a \boldsymbol{E}\ \mathit{ok} \\ \exists_{\leqslant 1} a \in \mathsf{dom}(\boldsymbol{A})\ .\ \exists Q, s, R\ .\ \boldsymbol{A}(a) = [Q\ \mathsf{MrdyA}(s\ R)]}{\Theta \vdash \boldsymbol{A}\ \mathit{ok}}$$

(Sys-T-EProg)

$$\frac{\Phi, \Delta \vdash \mathit{map}\ \mathit{ok} \quad \Phi, \Delta \vdash \mathsf{mesgQ}\ \mathit{ok} \quad \Phi, \Delta \vdash \boldsymbol{A}\ \mathit{ok} \quad \vdash \Phi\ \mathit{ok} \quad \mathsf{dom}(\boldsymbol{A}) = \mathsf{dom}(\mathit{map})}{\Phi \vdash \mathsf{eProg}(\Delta; [\mathit{map}\ \mathsf{mesgQ}]; \boldsymbol{A})\ \mathit{ok}}$$

Figure 7.3: System formation rules

Definition 7.2.2 (Valid System Context)
A located context Φ is a *valid system context*, written $\vdash \Phi\ ok$, if the following hold:

- Φ is closed, well-formed and does not clash with Φ_{aux}, ie. $\vdash_{\mathsf{L}} \Phi, \Phi_{aux}$;
- For any $x \in \mathsf{dom}(\Phi)$, $\Phi \vdash x \in \mathtt{Agent}^Z$ implies $\Phi(x) = \mathtt{Agent}^{\mathtt{s}}$; and
- For any $x \in \mathsf{dom}(\Phi)$, $\Phi \vdash x \in \hat{}^I T$ implies $\mathsf{containNoAgent}(T)$.

where $\mathsf{containNoAgent}(T)$ checks whether a type T contains occurrences of $\mathtt{Agent}^Z$, and is defined recursively as follows:

$$\begin{aligned}
\mathsf{containNoAgent}(\mathtt{Agent}^Z) &\stackrel{\text{def}}{=} \mathbf{false} \\
\mathsf{containNoAgent}(X) &\stackrel{\text{def}}{=} \mathbf{true} \\
\mathsf{containNoAgent}(B) &\stackrel{\text{def}}{=} \mathbf{true} \\
\mathsf{containNoAgent}(\mathtt{Site}) &\stackrel{\text{def}}{=} \mathbf{true} \\
\mathsf{containNoAgent}(\hat{}^I T) &\stackrel{\text{def}}{=} \mathsf{containNoAgent}(T) \\
\mathsf{containNoAgent}(\{X\} T) &\stackrel{\text{def}}{=} \mathbf{false} \\
\mathsf{containNoAgent}([T_1 \ldots T_n]) &\stackrel{\text{def}}{=} \mathsf{containNoAgent}(T_1) \wedge \ldots \wedge \mathsf{containNoAgent}(T_n)
\end{aligned}$$

These valid system contexts regulate the interactions between programs and their environment. As discussed in Section 5.4.2, observable agents are required to be static so that the standard definition of coupled simulation can be used (adding located type context indices). External channels (ie. channels on which input and output actions are observable) are prevented from sending or receiving agent names. This ensures that the daemon has a record of all agents in the system, as well as preventing programs from extruding mobile agent names to the environment (which would subsequently be able to observe their migration). This is not a severe restriction, as dynamically created new-bound agents and channels may have any type; moreover, we do not need to extrude the names of such agents. Additionally, since the interface context Φ_{aux} is binding in the top-level translation, we ensure that names of the system context do not clash with those in Φ_{aux}.

The typing rules for other forms of typing judgements are given in Figure 7.3. A brief informal description of the system formation rules follows. (Sys-T-Map) ensures the accuracy of the site map maintained by the daemon, and that it has a record of all agents, both those which are in the system context and those which are dynamically created and have been registered. Since the site map is represented as a list of pairs — possibly with duplicated entries — we require the accuracy of the consolidated list, with out-of-date entries removed.

(SYS-T-MESG) ensures all messages requested to be forwarded are well-typed. The rules (SYS-T-REGA), (SYS-T-MIGRATINGA) and (SYS-T-MIGRATEDA) check the typability of the pending process. The accuracy of current site, internally recorded by the agent, is ensured by (SYS-T-FREEA) and (SYS-T-REGA). (SYS-T-ASTATE) ensures typability of the process in the main body of each agent, as well as their pending state. Moreover, since the daemon is single-threaded, it ensures that the daemon is pending with at most one request. This means that there exists at most one agent whose pending state is $\mathsf{MrdyA}(s\ P)$, for some s, P. (SYS-T-EPROG) makes sure that each agent is well-formed, and that the domain of $\boldsymbol{A}$ is that of all declared agents in the system (so that, in sending inter-agent messages, target agents are always available).

Note that in a system $\mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$ which is well-formed w.r.t. Φ, there is much redundant location information: the site map map can be obtained from Φ, Δ, and also from the current site information locally recorded by each agent in $\boldsymbol{A}$. This allows the subject reduction result (Lemma 7.2.1) to state properties of the location information introduced by the encoding, eg. that the site map map is always accurate. Note further that, due to the implementation of finite map given in Section 6.5, when the site map map is updated, it will be appended with a new entry. This means that we need to maintain map as a list (with some duplicated entries) rather than a consolidated list, so that the subject reduction result (Lemma 7.2.1) can be stated.

7.2.2 Labelled Transition Rules

The labelled transitions of the intermediate language are of the form $\Phi \Vdash Sys \xrightarrow[\Delta]{\beta} Sys'$; their rules are given in Figures 7.5 to 7.8. The syntax of an IL label β is as that of $\mathrm{n}\pi_{\mathsf{LD,LI}}$. This means that the definitions of operational relations extend naturally to the intermediate language.

The semantics of IL uses structural congruence (in (SYS-EQUIV) in Figure 7.5). We define a structural congruence, indexed by a located type context Φ, $\equiv_\Phi$, to be the smallest relation between $\{Sys \mid \Phi \vdash Sys\ ok\}$, closed under the rules given in Figure 7.4 together with alpha-conversion of bound names. These rules allow extension of local scopes (SYS-STR-LOCAL-DEC), and rearrangement of top-level bindings, message queues and agent bodies ((SYS-STR-TOP-DEC), (SYS-STR-MESGQ-EQUIV) and (SYS-STR-LOCAL-EQUIV)).

The transition rules for the intermediate language are organised into three main categories, as given in the informal descriptions of the syntax, below.

$$\frac{a \notin \mathsf{dom}(\boldsymbol{A}) \quad \mathsf{dom}(\Theta) \cap \mathsf{dom}(\Phi, \Delta) = \emptyset}{\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}, a \mapsto [\texttt{new}\ \Theta\ \texttt{in}\ P\ \boldsymbol{E}]) \equiv_\Phi \mathsf{eProg}(\Delta, \Theta; \boldsymbol{D}; \boldsymbol{A}, a \mapsto [P\ \boldsymbol{E}])}\ (\textsc{Sys-Str-Local-Dec})$$

$$\frac{\Delta \equiv \Theta}{\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \equiv_\Phi \mathsf{eProg}(\Theta; \boldsymbol{D}; \boldsymbol{A})}\ (\textsc{Sys-Str-Top-Dec})$$

$$\frac{\mathsf{mesgQ} \equiv \mathsf{mesgQ}'}{\mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A}) \equiv_\Phi \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}']; \boldsymbol{A})}\ (\textsc{Sys-Str-MesgQ-Equiv})$$

$$\frac{\boldsymbol{A}(a) = [P\ \boldsymbol{E}] \quad P \equiv Q}{\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \equiv_\Phi \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [Q\ \boldsymbol{E}])}\ (\textsc{Sys-Str-Local-Equiv})$$

Figure 7.4: System structural congruence

$$\frac{Sys_1 \equiv_\Phi Sys_2 \quad \Phi \Vdash Sys_1 \xrightarrow[\Xi]{\beta} Sys_1' \quad Sys_1' \equiv_{\Phi,\Xi} Sys_2'}{\Phi \Vdash Sys_2 \xrightarrow[\Xi]{\beta} Sys_2'}\ (\textsc{Sys-Equiv})$$

Figure 7.5: System LTS: Closure

(Sys-Loc-Tau)

$$\frac{\begin{array}{l}\boldsymbol{A}(a) = [P|Q\ \boldsymbol{E}] \quad \Phi, \Delta \Vdash_a\ P \xrightarrow{\tau} @_a P' \\ P \equiv \textbf{if}\ v\ \textbf{then}\ P_1\ \textbf{else}\ P_2 \quad \vee\ P \equiv \textbf{let}\ p = ev\ \textbf{in}\ P_1\ \vee \\ P \equiv (c!\,v \mid c?p \rightarrow R) \qquad \vee\ P \equiv (c!\,v \mid *c?p \rightarrow R)\end{array}}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow{\tau} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [P'|Q\ \boldsymbol{E}])}$$

(Sys-Loc-Input)

$$\frac{\begin{array}{l}\{a, c\} \subseteq \mathsf{dom}(\Phi) \quad \boldsymbol{A}(a) = [(R \mid c?p \rightarrow Q)\ \boldsymbol{E}] \quad \mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta) = \emptyset \\ \Phi \vdash c \in \hat{}^{\mathbf{r}}T \quad \Phi, \Xi \vdash v \in T \quad \mathsf{dom}(\Xi) \subseteq \mathsf{fv}(v) \quad \Xi \text{ extensible}\end{array}}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow[\Xi]{@_a c?v} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [(P|\mathsf{match}(p, v)Q)\ \boldsymbol{E}])}$$

(Sys-Loc-Replic)

$$\frac{\begin{array}{l}\{a, c\} \subseteq \mathsf{dom}(\Phi) \quad \boldsymbol{A}(a) = [(R \mid *c?p \rightarrow Q)\ \boldsymbol{E}] \quad \mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta) = \emptyset \\ \Phi \vdash c \in \hat{}^{\mathbf{r}}T \quad \Phi, \Xi \vdash v \in T \quad \mathsf{dom}(\Xi) \subseteq \mathsf{fv}(v) \quad \Xi \text{ extensible}\end{array}}{\begin{array}{l}\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \\ \quad \xrightarrow[\Xi]{@_a c?v} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [(P \mid \mathsf{match}(p, v)Q \mid *c?p \rightarrow Q)\ \boldsymbol{E}])\end{array}}$$

(Sys-Loc-Output)

$$\frac{\begin{array}{l}\{a, c\} \subseteq \mathsf{dom}(\Phi) \quad \boldsymbol{A}(a) = [(P|c!\,v)\ \boldsymbol{E}] \\ \Delta \equiv \Delta_1, \Delta_2 \quad \mathsf{dom}(\Delta) \cap \mathsf{fv}(v) = \mathsf{dom}(\Delta_1)\end{array}}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow[\Delta_1]{@_a c!v} \mathsf{eProg}(\Delta_2; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [P\ \boldsymbol{E}])}$$

(Sys-Loc-IfLocal-T)

$$\frac{\begin{array}{l}\boldsymbol{A}(a) = [P|Q\ \boldsymbol{E}] \quad \boldsymbol{A}(b) = [R\ \boldsymbol{E}'] \quad P = \textbf{iflocal}\ \langle b \rangle c!\,v\ \textbf{then}\ P_1\ \textbf{else}\ P_2 \\ \Phi, \Delta \vdash a@s \quad \Phi, \Delta \vdash b@s \quad a \neq b\end{array}}{\begin{array}{l}\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \\ \quad \xrightarrow{\tau} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [P_1|Q\ \boldsymbol{E}] \oplus b \mapsto [c!\,v|R\ \boldsymbol{E}'])\end{array}}$$

(Sys-Loc-IfLocal-Same)

$$\frac{\boldsymbol{A}(a) = [P|Q\ \boldsymbol{E}] \quad P = \textbf{iflocal}\ \langle a \rangle c!\,v\ \textbf{then}\ P_1\ \textbf{else}\ P_2}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow{\tau} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [c!\,v|P_1|Q\ \boldsymbol{E}])}$$

(Sys-Loc-IfLocal-F)

$$\frac{\begin{array}{l}\boldsymbol{A}(a) = [P|Q\ \boldsymbol{E}] \quad P = \textbf{iflocal}\ \langle b \rangle c!\,v\ \textbf{then}\ P_1\ \textbf{else}\ P_2 \\ \Phi, \Delta \vdash a@s \quad \Phi, \Delta \vdash b@s' \quad s \neq s'\end{array}}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow{\tau} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [P_2|Q\ \boldsymbol{E}])}$$

Figure 7.6: System LTS: Local computation

- Local Computation (Sys-Loc-*): The rules in this category are given in Figure 7.6. In (Sys-Loc-Tau), an agent performs an internal computation (including channel communication). In (Sys-Loc-Input), (Sys-Loc-Replic) and (Sys-Loc-Output), an agent interacts with the environment via an input or output action, extruding private names if necessary. LD messaging using **iflocal** may move an output to another agent. This is dealt with by (Sys-IfLocal-T) when LD messaging succeeds, by (Sys-IfLocal-Same) if the target agent is the same as the executing agent, and (Sys-IfLocal-F) otherwise.

(Sys-Req-Reg)

$$\frac{\boldsymbol{A}(a) = [((\mathbf{create}^{Z}\ b = P\ \mathbf{in}\ Q) \mid R)\ \mathsf{FreeA}(s)] \quad b \notin \mathsf{dom}(\Phi, \Delta)}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow{\tau} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)])}$$

(Sys-Req-Mig)

$$\frac{\boldsymbol{A}(a) = [((\mathbf{migrate\ to}\ s \to P) \mid Q)\ \mathsf{FreeA}(s')]}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow{\tau} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [Q\ \mathsf{MtingA}(s\ P)])}$$

(Sys-Req-Mesg)

$$\frac{\boldsymbol{A}(a) = [(\langle b@?\rangle c!v \mid P)\ \boldsymbol{E}] \quad (\Phi, \Delta)(c) = \hat{\ }^{I}T}{\begin{array}{l}\Phi \Vdash \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A}) \\ \quad \xrightarrow{\tau} \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}|\mathsf{mesgReq}(\{T\}\ [b\ c\ v])]; \boldsymbol{A} \oplus a \mapsto [P\ \boldsymbol{E}])\end{array}}$$

Figure 7.7: System LTS: Initialising request

- Initialising Requests (Sys-Req-*): The rules in this category are given in Figure 7.7. Creations and migrations may initialise if the local lock of the agent is free, ie. if the agent is in a FreeA(s) state ((Sys-Req-Reg) and (Sys-Req-Mig)). (Sys-Req-Mesg), however, allows initialisation of LI messaging regardless of the local lock's state.

- Request Processing (Sys-Proc-*) and (Sys-Comm-Mig): if the daemon is idle, one of the daemon requests may proceed (Sys-Proc-*). The rules in this category are given in Figure 7.8. If the daemon is busy, it may finish its pending request and return to an idle state (Sys-Comm-Mig). The operational semantics precisely captures the informal description given previously.

As in the operational semantics of Nomadic π, the semantics is typed, ie. $\xrightarrow[\Delta]{\beta}$ is the smallest relation from $\{\Phi \Vdash Sys \mid \Phi \vdash Sys\ ok \wedge\ \vdash \Phi, \Delta\ ok\}$ to $\{\Phi' \Vdash Sys' \mid Sys' \in \mathrm{IL}\}$. This avoids having to deal with systems containing ill-typed processes. More specifically to the

(SYS-PROC-REG)

$$\frac{\boldsymbol{A}(a) = [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)] \quad b \notin \mathsf{dom}(\Phi, \Delta) \qquad \mathsf{eProg}(\Delta; [\mathit{map}\ \mathsf{mesgQ}]; \boldsymbol{A})\ \text{idle}}{\begin{array}{l}\Phi \Vdash \mathsf{eProg}(\Delta; [\mathit{map}\ \mathsf{mesgQ}]; \boldsymbol{A}) \\ \quad \xrightarrow{\tau}\ \mathsf{eProg}(\Delta,\ b : \mathtt{Agent}^{Z}@s; [[b\ s]\mathbf{::}\mathit{map}\ \mathsf{mesgQ}]; \\ \qquad\qquad \boldsymbol{A} \oplus a \mapsto [Q|R\ \mathsf{FreeA}(s)] \oplus b \mapsto [P\ \mathsf{FreeA}(s)])\end{array}}$$

(SYS-PROC-MIG)

$$\frac{\boldsymbol{A}(a) = [R\ \mathsf{MtingA}(s\ P)] \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})\ \text{idle}}{\Phi \Vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}) \xrightarrow{\tau} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [R\ \mathsf{MrdyA}(s\ P)])}$$

(SYS-COMM-MIG)

$$\frac{\boldsymbol{A}(a) = [R\ \mathsf{MrdyA}(s\ P)]}{\begin{array}{l}\Phi \Vdash \mathsf{eProg}(\Delta; [\mathit{map}\ \mathsf{mesgQ}]; \boldsymbol{A}) \\ \quad \xrightarrow{\tau} \mathsf{eProg}(\Delta \oplus a \mapsto s; [[a\ s]\mathbf{::}\mathit{map}\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a \mapsto [P|R\ \mathsf{FreeA}(s)])\end{array}}$$

(SYS-PROC-MESG)

$$\frac{\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})\ \text{idle} \quad \boldsymbol{A}(a) = [P\ \boldsymbol{E}]}{\begin{array}{l}\Phi \Vdash \mathsf{eProg}(\Delta; [\mathit{map}\ \mathsf{mesgQ}|\mathsf{mesgReq}(\{T\}\ [a\ c\ v])]; \boldsymbol{A}) \\ \quad \xrightarrow{\tau} \mathsf{eProg}(\Delta; [\mathit{map}\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a \mapsto [c\mathbf{!}v|P\ \boldsymbol{E}])\end{array}}$$

Figure 7.8: System LTS: Request processing

intermediate language, the typed semantics ensures that migrating agents are always bound at top-level, and that target agents are always available in inter-agent messaging. The type system also ensures certain invariants (eg. correctness of the site map and of each agent's internally recorded current site), and therefore situations such as a "halting" daemon (due to a failed agent look up) never occur. A subject reduction result, stated below, ensures well-formedness of systems is indeed preserved by transitions.

Lemma 7.2.1 (Subject reduction for IL)
Given a valid system context Φ, if $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ then $\Phi, \Xi \vdash Sys'\ ok$.

Proof: An induction on the transition derivation. See Appendix D.1 on page 245 for details. ∎

Note that we exclude cases where the extruded type context clashes with names in the interface context Φ_{aux}, which defines the names used in the $\mathcal{C}$-translation. Since Φ_{aux} is non-sendable in $\mathcal{C}[\![LP]\!]$, no part of Φ_{aux} will be extruded by an output action. The next lemma shows that the restrictions on interactions between the program and its environment, regulated by valid system contexts, are respected.

Lemma 7.2.2 (Extrusion/intrusion of names)
Given that $\Phi \vdash Sys\ ok$ and $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$, the following hold:

- Ξ only contains channel names;
- $\beta \neq @_a \mathsf{migrate\ to}\ s$ for any a, s; and
- $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ implies $\vdash \Phi, \Xi\ ok$.

Proof: We prove that Ξ only contains channel names by an induction on the transition derivation. Proving that β is never a migration action is easy — there is no rule that permits such an action. To prove the last statement, we observe that if $x \in \mathsf{dom}(\Xi)$ and $\Xi(x) = T$ then $\mathsf{containNoAgent}(T)$ is valid. ∎

7.3 Factorisation of $\mathcal{C}$-Encoding

In this section we factorise the $\mathcal{C}$-translation into the composition of two maps: a *loading* encoding $\mathcal{L}$ and a *flattening* encoding $\mathcal{F}$. For clarity, we repeat the diagram below, which outlines the relationship between the encodings and the languages.

$$\begin{tikzcd} \mathrm{n}\pi_{\mathsf{LD,LI}} \arrow[r, "\mathcal{L}[\![\cdot]\!]"] \arrow[dr, "\mathcal{C}[\![\cdot]\!]"'] & \mathrm{IL} \arrow[d, "\mathcal{F}[\![\cdot]\!]"] \\ & \mathrm{n}\pi_{\mathsf{LD}} \end{tikzcd}$$

We prove properties of these encodings and, in particular, investigate the behavioural relationship between the intermediate language and the target language. The major result shown is the correctness of the intermediate language, ie. that $\mathcal{F}$ is an expansion (Lemma 7.3.5).

7.3.1 Loading

The *loading* encoding $\mathcal{L} : \mathrm{n}\pi_{\mathsf{LD,LI}} \to \mathrm{IL}$ is defined for any LP well-typed w.r.t. a valid system context Φ. It uses the compositional encoding $[\![\cdot]\!]$ for located process, defined in Section 2.3 on page 32, for rearranging a located process LP into the form

$$\texttt{new}\ \Delta\ \texttt{in}\ \prod_i @_{a_i} P_i$$

where all a_i are distinct. Such a located process is mapped to a system

$$\mathsf{eProg}(\Delta; [\mathsf{Enlist}(\Phi,\Delta)\ \bullet]; \boldsymbol{A})$$

where each agent a_i of $\boldsymbol{A}$ is initialised with state $[P_i\ \mathsf{FreeA}(s_i)]$ (s_i is the current site of a_i in the context). If an agent a is declared in the external context Φ or Δ, yet does not locate any process in LP, it has to be initialised with $\mathbf{0}$ as its main body (this is required by (Sys-T-EProg)). The definition of the loading encoding is given below.

$$\begin{aligned} \mathcal{L}_\Phi [\![LP]\!] &\stackrel{\text{def}}{=} \mathsf{eProg}(\Delta; [\mathsf{Enlist}(\Phi,\Delta)\ \bullet]; \boldsymbol{A}) \\ &\qquad \text{where } \boldsymbol{A} = \mathsf{loadLP}_{\Phi,\Delta}(\mathsf{A}) \ \wedge\ [\![LP]\!] = (\Delta; \mathsf{A}) \end{aligned}$$

$$\mathsf{loadLP}_{\Phi,\Delta}(\mathsf{A})(a) \stackrel{\text{def}}{=} \begin{cases} [\mathsf{A}(a)\ \mathsf{FreeA}(s)] & \Phi,\Delta \vdash a@s \ \wedge\ a \in \mathsf{dom}(\mathsf{A}) \\ [\mathbf{0}\ \mathsf{FreeA}(s)] & \Phi,\Delta \vdash a@s \ \wedge\ a \notin \mathsf{dom}(\mathsf{A}) \end{cases}$$

We have proved that this loading encoding results in a well-formed system.

Lemma 7.3.1 (Loading preserves typing)
If $\Gamma \vdash LP$ and Γ is a valid system context then $\Gamma \vdash \mathcal{L}[\![LP]\!]\ ok$.

Proof: Assume, without lost of generality, that

$$LP \equiv \texttt{new}\ \Delta\ \texttt{in}\ (@_{a_1} P_1 \mid \ \ldots\ \mid @_{a_n} P_n)$$

where $a_1 \ldots a_n$ are distinct; moreover $\Gamma \vdash a_i@s_i$, for each i. By definition, we have:

$$\mathcal{L}_\Gamma [\![LP]\!] \quad = \quad \mathsf{eProg}(\Delta; [\mathsf{Enlist}(\Gamma, \Delta)\ \bullet]; \boldsymbol{A})$$

where $\boldsymbol{A}$ is defined in such a way that $\boldsymbol{A}(a_i) = [P_i\ \mathsf{FreeA}(s_i)]$.

Since $\Gamma \vdash LP$ and $\vdash \Gamma\ ok$, we have the following.

- $\Gamma, \Delta \vdash_{a_i} P_i$ for all i, by (New) and (Prl). This means that $\Gamma \vdash \boldsymbol{A}\ ok$, by (Sys-T-AState).
- $\Gamma, \Delta \vdash \mathsf{Enlist}(\Gamma, \Delta)\ ok$ and $\mathsf{dom}(\mathsf{Enlist}(\Gamma, \Delta)) = \mathsf{dom}(\Gamma, \Delta)$, by construction.

Hence $\Gamma \vdash \mathsf{eProg}(\Delta; [\mathsf{Enlist}(\Gamma, \Delta)\ \bullet]; \boldsymbol{A})$, by (Sys-T-EProg). ■

7.3.2 Flattening

The *flattening* function $\mathcal{F} : \mathrm{IL} \to \mathrm{n}\pi_{\mathsf{LD}}$ maps well-formed systems to corresponding processes in the target language. Basically a flattened system consists of two parts: the flattened state of the daemon, and that of agents. Flattening the daemon involves installing a *Daemon* process and its message queue mesgQ, a parallel composition of $\mathsf{mesgReq}$ processes, at D. Moreover, we need to restore the daemon lock according to the status of the system. If the system is idle, `lock` is simply outputting the site map m (constructed from the list *map* using $\mathsf{makeMap}(m; map)$). Otherwise, if the daemon is serving a migration request of an agent a, it is actually waiting for an acknowledgement on a fresh `migrated` channel. In this case, the target-level status of the daemon and the agent are $\mathsf{migProc}(a\ m\ \mathtt{migrated})$ and $\mathsf{migReady}(s\ \mathtt{migrated}\ Q)$ respectively (see Figures 2.3 and 2.4 for their definitions). Restoring the state involving the daemon lock is done by the *site map flattening function* $\mathsf{mapS}(\cdot)$, formally defined below. Note that the scope of `migrated` is over the daemon as well as a, and therefore the state of agent a must be included as an argument of such a function.

$$\begin{array}{lcl}
\mathsf{mapS}_\bullet(map) & \stackrel{\mathrm{def}}{=} & @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in} \\
 & & \quad (\mathtt{lock!}m \mid \mathsf{makeMap}(m; map)) \\
\mathsf{mapS}_{(a,\ [P\ \mathsf{MrdyA}(s\ Q)])}(map) & \stackrel{\mathrm{def}}{=} & \mathbf{new}\ \mathtt{migrated} : \hat{}^{\mathtt{rw}}[\mathtt{Site}\ \hat{}^{\mathtt{w}}[]]\ \mathbf{in} \\
 & & \quad @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in} \\
 & & \quad\quad (\mathsf{makeMap}(m; map) \mid \mathsf{migProc}(a\ m\ \mathtt{migrated})) \\
 & & \quad \mid @_a(\mathsf{migReady}(s\ \mathtt{migrated}\ Q) \mid [\![P]\!]_a \mid \mathtt{Deliverer})
\end{array}$$

Flattening the state $\boldsymbol{A}$ of the agents results in a parallel composition of flattened agents. To flatten an agent a, its main body is put in a normal form by the compositional encoding $[\![]\!]$ of the example infrastructure. Such an agent is located at a, together with a `Deliverer` and a flattened pending state. If the pending state is $\mathsf{FreeA}(s)$ (ie. there is no pending process), flattening introduces an output $\mathtt{currentloc!}s$. In case $\mathsf{RegA}(b\ Z\ s\ P\ Q)$, the new agent b has already been created, together with fresh channels `pack` and `rack`, used for acknowledgements from the daemon to the new agent and from the new agent to its parent respectively. Furthermore, the *Daemon* process at D has already reacted to the registration message, spawning $\mathsf{regReq}(b\ s\ \mathtt{rack})$. Meanwhile b and its parent are waiting for an acknowledgement before resuming execution. Their target-level status are therefore $\mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P)$ and $\mathsf{regBlockP}(s\ \mathtt{pack}\ Q)$ respectively. Similarly for $\mathsf{MtingA}(s\ P)$, the migrating message reacts with *Daemon*, producing $\mathsf{migReq}(a\ \mathtt{mack})$, with `mack` being a fresh channel. The migrating agent itself waits for a go-ahead signal from the daemon via `mack`, hence its target-level status is $\mathsf{migBlock}(s\ \mathtt{mack}\ P)$.

$$\mathcal{F}[\![\boldsymbol{A}]\!] \stackrel{\text{def}}{=} \prod_{a\in\mathsf{dom}(\boldsymbol{A})} \mathcal{F}[\![\boldsymbol{A}(a)]\!]_a$$

$$\mathcal{F}[\![[P\ \boldsymbol{E}]]\!]_a \stackrel{\text{def}}{=} \mathcal{F}[\![\boldsymbol{E}]\!]_a \mid @_a([\![P]\!]_a \mid \mathtt{Deliverer})$$

$$\begin{aligned}
\mathcal{F}[\![\mathsf{FreeA}(s)]\!]_a &\stackrel{\text{def}}{=} @_a\mathtt{currentloc!}s \\
\mathcal{F}[\![\mathsf{RegA}(b\ Z\ s\ P\ Q)]\!]_a &\stackrel{\text{def}}{=} \mathbf{new}\ \mathtt{pack} : \mathtt{\hat{}}^{\mathtt{rw}}[],\ \mathtt{rack} : \mathtt{\hat{}}^{\mathtt{rw}}[],\ b : \mathtt{Agent}^Z@s\ \mathbf{in} \\
&\qquad @_D\mathsf{regReq}(b\ s\ \mathtt{rack}) \\
&\quad\ \mid @_a\mathsf{regBlockP}(s\ \mathtt{pack}\ Q) \\
&\quad\ \mid @_b\mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P) \\
\mathcal{F}[\![\mathsf{MtingA}(s\ P)]\!]_a &\stackrel{\text{def}}{=} \mathbf{new}\ \mathtt{mack} : \mathtt{\hat{}}^{\mathtt{rw}}[\mathtt{\hat{}}^{\mathtt{w}}[\mathtt{Site}\ \mathtt{\hat{}}^{\mathtt{w}}[]]]\ \mathbf{in} \\
&\quad\ @_D\mathsf{migReq}(a\ \mathtt{mack}) \mid @_a\mathsf{migBlock}(s\ \mathtt{mack}\ P)
\end{aligned}$$

Flattening a system $Sys = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$ is defined below. We implicitly use alpha-conversion so that $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$.

$$\mathcal{F}[\![Sys]\!] \stackrel{\text{def}}{=} \begin{cases}
\begin{aligned}&\mathbf{new}\ \Delta, \Phi_{aux}\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \\ &\quad \mid \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}]\!])\end{aligned} & Sys \text{ is idle} \\
\begin{aligned}&\mathbf{new}\ \Delta, \Phi_{aux}\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \\ &\quad \mid \mathsf{mapS}_{(a,\ [P\ \mathsf{MrdyA}(s\ Q)])}(map) \mid \mathcal{F}[\![\boldsymbol{A}']\!])\end{aligned} & \begin{aligned}&\boldsymbol{A}(a) = [P\ \mathsf{MrdyA}(s\ Q)] \wedge \\ &\boldsymbol{A}' = \boldsymbol{A}/\{a\}\end{aligned}
\end{cases}$$

Unless the pending state is $\mathsf{FreeA}(s)$, an agent is partially committed. Its translation therefore does not correspond to any agent in the image of the $\mathcal{C}$-translation. Only the translation of a fully-committed system (ie. one with each agent in a $\mathsf{FreeA}(s)$ state, such as that obtained from loading) corresponds to $\mathcal{C}[\![LP]\!]$ for some LP. If we load and then flatten a (high-level) located process LP, we obtain its translation $\mathcal{C}[\![LP]\!]$.

Lemma 7.3.2 (Factorisation)
If $\Phi \vdash LP$ then $\mathcal{F}[\![\mathcal{L}_\Phi[\![LP]\!]]\!] \equiv \mathcal{C}_\Phi[\![LP]\!]$.

Proof: Assume, without lost of generality, that

$$LP \equiv \mathtt{\textbf{new}}\ \Delta\ \mathtt{\textbf{in}}\ (@_{a_1}P_1 \mid \ \ldots \ \mid @_{a_n}P_n)$$

where $a_1 \ldots a_n$ are distinct; moreover, $\Phi \vdash a_i@s_i$ for all i. By definition, we have:

$$\mathcal{L}_\Phi[\![LP]\!] \quad = \quad \mathsf{eProg}(\Delta; [\mathsf{Enlist}(\Phi, \Delta)\ \bullet]; \boldsymbol{A})$$

where $\boldsymbol{A}$ is defined so that $\boldsymbol{A}(a_i) = [P_i\ \mathsf{FreeA}(s_i)]$. We have the following.

- Since $\mathcal{L}_\Phi[\![LP]\!]$ is idle, we have:

$$\mathcal{F}[\![\mathcal{L}_\Phi[\![LP]\!]]\!] \quad = \quad \mathtt{\textbf{new}}\ \Delta, \Phi_{aux}\ \mathtt{\textbf{in}}\ (@_D Daemon \mid \mathsf{mapS}_\bullet(\mathsf{Enlist}(\Phi, \Delta)) \mid \mathcal{F}[\![\boldsymbol{A}]\!]).$$

- $\mathcal{F}[\![\boldsymbol{A}]\!] \equiv \prod_{i \in 1 \ldots n}(@_{a_i}([\![P_i]\!]_{a_i} \mid \mathtt{currentloc!}s_i \mid \mathtt{Deliverer}))$, by definition.
- $\mathsf{mapS}_\bullet(\mathsf{Enlist}(\Phi, \Delta)) = \ @_D\mathtt{\textbf{new}}\ m : \mathtt{Map}[\mathtt{Agent^s\ Site}]\ \mathtt{\textbf{in}}$ $(\mathtt{lock!}m \mid \mathsf{makeMap}(m; \mathsf{Enlist}(\Phi, \Delta)))$.

Clearly $\mathcal{F}[\![\mathcal{L}_\Phi[\![LP]\!]]\!] \equiv \mathcal{C}_\Phi[\![LP]\!]$. Hence the lemma. ∎

7.3.3 Behavioural Properties

We wish to show the correctness of our intermediate language, ie. that each of its terms is behaviourally equivalent to some target-level term. To prove such a result, we first need to establish operational correspondence results between a system in IL and its flattened target term. If an action is observable, the transition of an intermediate language term corresponds exactly with that of its flattened image in the target language. This is also true in the cases where the action is caused by local computation. Cases involving house-keeping steps, however, are more complex. Each flattened IL term (which is in $\mathrm{n}\pi_{\mathsf{LD}}$) is stable in the sense that it cannot perform any house-keeping τ-step introduced by the $\mathcal{C}$-encoding (although it may perform some partial-commitment steps). Supposing a flattened system $\mathcal{F}[\![Sys]\!]$ reduces to a located process LP, where such a reduction is introduced by the $\mathcal{C}$-encoding. This process does not correspond to any flatten of system, since LP is capable of performing some house-keeping steps. However, Lemma 7.3.4 ensures that such house-keeping steps induce expansions, and therefore LP can be related by an expansion to a flattened system $\mathcal{F}[\![Sys']\!]$, where Sys' is the result of the corresponding reduction of Sys, matching that of $\mathcal{F}[\![Sys]\!]$ to LP.

Lemma 7.3.3 (Completeness of IL)
Given a valid system context Φ, if $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ then there exists LP' such that $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow[\Xi]{\beta} LP'$, and $LP' \dot{\succeq}^{\emptyset}_{\Phi,\Xi} \mathcal{F}[\![Sys']\!]$.

Lemma 7.3.4 (Soundness of IL)
Given a valid system context Φ, if $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow[\Xi]{\beta} LP$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ then there exists Sys' such that $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$, and $LP \dot{\succeq}^{\emptyset}_{\Phi,\Xi} \mathcal{F}[\![Sys']\!]$.

The proofs of the lemmas above use induction on the transition derivation. As in the subject reduction result (Lemma 7.2.1), we exclude cases where extruded type context clashes with Φ_{aux}. The congruence properties of translocating expansion are heavily used, for factoring out program contexts which are not involved in house-keeping reductions of the target terms. Temporary immobility is used whenever we need to guarantee that LD messages to partially committed agents are safely delivered. We give a proof sketch for Lemma 7.3.3 below. The details are in Appendix D.2 on page 249.

Proof Sketch: Consider $Sys = \mathsf{eProg}(\Delta'; \boldsymbol{D}'; \boldsymbol{A}')$ well-formed wrt Φ. Since $\mathcal{F}$ is only defined if $\mathsf{dom}(\Delta') \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, we pick an injective substitution $\sigma : \mathsf{dom}(\Delta') \to \mathcal{X}/\mathsf{dom}(\Phi, \Phi_{aux})$ and denote $\Delta = \sigma\Delta'$, $\boldsymbol{D} = \sigma\boldsymbol{D}$ and $\boldsymbol{A}' = \sigma\boldsymbol{A}$. Clearly we have $Sys \stackrel{\alpha}{=} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ and, by (L-C-Var), $\vdash \Phi, \Phi_{aux}, \Delta$. By this alpha-conversion, $\mathcal{F}[\![Sys]\!]$ is defined. We demonstrate the case where $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ and (Sys-Proc-Mig) is the only rule used for deriving this transition.

Case (Sys-Proc-Mig): Supposing $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ is idle and $\boldsymbol{A}(a_e) = [Q\ \mathsf{MtingA}(s\ P)]$. By (Sys-Proc-Mig), we have $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{MrdyA}(s\ P)])$$

Let $\boldsymbol{A}'$ is obtained by excluding a_e from the domain of $\boldsymbol{A}$ and $\mathcal{E}[\cdot], LQ$ be defined as follows:

$$\begin{aligned}
\mathcal{E}[\cdot] &= \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ (@_D(\mathit{Daemon} \mid \mathsf{mesgQ}) \mid [\cdot] \mid \mathcal{F}[\![\boldsymbol{A}']\!]) \\
LQ &= \mathsf{mapS}_{\bullet}(\mathit{map}) \mid \mathcal{F}[\![\boldsymbol{A}(a_e)]\!]_{a_e} \\
&= \quad @_D(\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathtt{lock!}m \mid \mathsf{makeMap}(m; \mathit{map}))) \\
&\quad\ \mid @_{a_e}([\![Q]\!]_{a_e} \mid \mathtt{Deliverer}) \\
&\quad\ \mid \mathbf{new}\ \mathtt{mack} : \texttt{\^{}rw}[\texttt{\^{}w}[\mathtt{Site}\ \texttt{\^{}w}[]]]\ \mathbf{in}\ (@_D\mathsf{migReq}(a_e\ \mathtt{mack}) \mid @_{a_e}\mathsf{migBlock}(s\ \mathtt{mack}\ P))
\end{aligned}$$

$$
\begin{array}{ll}
\equiv & \mathbf{new}\ \mathtt{mack} : \hat{\ }^{\mathtt{rw}}[\hat{\ }^{\mathtt{w}}[\mathtt{Site}\ \hat{\ }^{\mathtt{w}}[]]]\ \mathbf{in} \\
& \quad (@_D(\mathsf{migReq}(a_e\ \mathtt{mack}) \\
& \qquad \mid \mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathtt{lock!}m \mid \mathsf{makeMap}(m; map))) \\
& \quad \mid @_{a_e}(\mathsf{migBlock}(s\ \mathtt{mack}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer}))
\end{array}
$$

Clearly, $\mathcal{F}[\![Sys]\!] \equiv \mathcal{E}[LQ]$. In this case, we have:

- $\mathcal{E}[\Phi] \Vdash LQ \xrightarrow{\tau} LQ'$, by (Lts-L-Out), (Lts-L-In) and (Lts-L-Comm), (Lts-Prl) and (Lts-New), where

$$
\begin{array}{lll}
LQ' & \equiv & \mathbf{new}\ \mathtt{mack} : \hat{\ }^{\mathtt{rw}}[\hat{\ }^{\mathtt{w}}[\mathtt{Site}\ \hat{\ }^{\mathtt{w}}[]]]\ \mathbf{in} \\
& & \quad (LR \mid @_{a_e}(\mathsf{migBlock}(s\ \mathtt{mack}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer})) \\
LR & = & @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathsf{makeMap}(m; map) \\
& & \quad \mid \mathbf{lookup}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ a_e\ \mathbf{in}\ m\ \mathbf{with} \\
& & \qquad \mathbf{found}(s') \rightarrow \mathbf{new}\ \mathtt{migrated} : \hat{\ }^{\mathtt{rw}}[\mathtt{Site}\ \hat{\ }^{\mathtt{rw}}[]]\ \mathbf{in} \\
& & \qquad\quad (\langle a_e @ s'\rangle \mathtt{mack!}[\mathtt{migrated}] \mid \mathsf{migProc}(a_e\ \mathtt{migrated})) \\
& & \qquad \mathbf{notfound} \rightarrow \mathbf{0})
\end{array}
$$

- $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow{\tau} LP'$, by (Lts-Prl) and (Lts-New), where $LP' = \mathcal{E}[LQ']$.

We now need to verify that $LP' \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{F}[\![Sys']\!]$. Let $M = \mathsf{mov}(\mathcal{E}[\Phi])/\{a_e\}$.

- LR may look up a_e in the site map; this step yields an expansion. By Lemma 6.5.3, let $\mathsf{lookupL}(a_e; map) = s_e$, we have: $LR \succeq_{\mathcal{E}[\Phi]} LR_1$ where

$$
\begin{array}{lll}
LR_1 & \equiv & @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathsf{makeMap}(m; map) \\
& & \quad \mid \mathbf{new}\ \mathtt{migrated} : \hat{\ }^{\mathtt{rw}}[\mathtt{Site}\ \hat{\ }^{\mathtt{rw}}[]]\ \mathbf{in} \\
& & \qquad (\langle a_e @ s_e\rangle \mathtt{mack!}[\mathtt{migrated}] \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})))
\end{array}
$$

  This means, by Theorem 5.4.3, $LQ' \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} LQ_1$ where

$$
\begin{array}{lll}
LQ_1 & \equiv & \mathbf{new}\ \mathtt{mack} : \hat{\ }^{\mathtt{rw}}[\hat{\ }^{\mathtt{w}}[\mathtt{Site}\ \hat{\ }^{\mathtt{w}}[]]]\ \mathbf{in} \\
& & \quad (LR_1 \mid @_{a_e}(\mathsf{migBlock}(s\ \mathtt{mack}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer}))
\end{array}
$$

- We now need to redefine the scope of `currentloc` to use the temporary immobility result to show that $\langle a_e @ s_e\rangle \mathtt{mack!}[\mathtt{migrated}]$ will be safely delivered to a_e. We define the following.

$$
\begin{array}{lll}
\mathcal{E}'[\cdot] & = & \mathbf{new}\ \Phi^{-}_{aux}, \Delta\ \mathbf{in}\ @_D(Daemon \mid \mathsf{mesgQ}) \mid [\cdot] \\
& & \quad \mid \mathbf{new}\ \mathtt{currentloc} : \hat{\ }^{\mathtt{rw}}\mathtt{Site}\ \mathbf{in}\ \mathcal{F}[\![A']\!] \\
\Phi^{-}_{aux} & \stackrel{\text{def}}{=} & \Phi_{aux}/\mathtt{currentloc} : \hat{\ }^{\mathtt{rw}}\mathtt{Site}
\end{array}
$$

Since `currentloc` is a local channel, applying Lemma 6.2.7, we have:

$$\mathcal{E}[LQ_1] \quad \dot{\sim}^{\emptyset}_{\Phi} \quad \mathcal{E}'[\mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{~}^{rw}Site}\ \mathbf{in}\ LQ_1]$$

- By (Sys-T-Map), $a_e \in \mathsf{dom}(map)$, this means there exists s_e such that

$$\mathsf{lookupL}(a_e; map) = s_e$$

moreover, $\Phi, \Delta \vdash a_e @ s_e$, by Lemma D.1.4. This implies

$$\begin{aligned}
&\mathcal{E}[\Phi],\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]} \Vdash LR_1 \xrightarrow[M]{\mathsf{det}} LR_2 \quad \text{where} \\
&LR_2 \ = \ \mathbf{new}\ \mathtt{migrated} : \mathtt{\hat{~}^{rw}[Site\ \hat{~}^{rw}[]]}\ \mathbf{in} \\
&\qquad (@_D\mathbf{new}\ m : \mathtt{Map[Agent^{s}\ Site]}\ \mathbf{in} \\
&\qquad\quad (\mathsf{makeMap}(m; map) \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})) \\
&\qquad \mid @_{a_e}\mathtt{mack!}[\mathtt{migrated}])
\end{aligned}$$

- By Lemma 6.4.5, the process below (which shall be denoted by LR') is temporary immobile blocked by `mack` w.r.t. $(\mathcal{E}[\Phi],\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}), M$.

$$\mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{~}^{rw}Site}\ \mathbf{in}\ @_{a_e}(\mathsf{migBlock}(s\ \mathtt{mack}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer})$$

Applying Lemma 6.4.2, we have

$$\begin{aligned}
&\mathbf{new}\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in}\ (LR_1 \mid LR') \\
&\quad \dot{\succeq}^{M}_{\mathcal{E}'[\Phi]}\ \mathbf{new}\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in}\ (LR_2 \mid LR')
\end{aligned}$$

- Since $a_e \notin \mathsf{mayMove}(\mathcal{F}[\![\boldsymbol{A}']\!], @_D \ldots)$, by Theorem 5.4.3,

$$\begin{aligned}
&\mathcal{E}'[\mathbf{new}\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in}\ LR_1 \mid LR'] \\
&\quad \dot{\succeq}^{\emptyset}_{\Phi}\ \mathcal{E}'[\mathbf{new}\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in}\ LR_2 \mid LR']
\end{aligned}$$

Moreover, since

$$\begin{aligned}
&\mathbf{new}\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in}\ (LR_1 \mid LR') \\
&\quad \equiv \mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{~}^{rw}Site}\ \mathbf{in}\ LQ_1 \qquad \text{and} \\
&\mathbf{new}\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in}\ (LR_2 \mid LR') \\
&\quad \equiv \mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{~}^{rw}Site},\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in} \\
&\qquad\quad (LR_2 \mid @_{a_e}(\mathsf{migBlock}(s\ \mathtt{mack}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer})) \\
&\quad \equiv \mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{~}^{rw}Site}\ \mathbf{in}\ LQ_2 \qquad \text{where} \\
&LQ_2 \ = \ \mathbf{new}\ \mathtt{mack} : \mathtt{\hat{~}^{rw}[\hat{~}^{w}[Site\ \hat{~}^{w}[]]]}\ \mathbf{in} \\
&\qquad\quad (LR_2 \mid @_{a_e}(\mathsf{migBlock}(s\ \mathtt{mack}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer}))
\end{aligned}$$

Since `currentloc` is a local channel, applying Lemma 6.2.7, we have: $\mathcal{E}[LQ_1] \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{E}[LQ_2]$.

- Since `migrated` is a linear channel, a communication on such a channel yields an expansion. By Fact 6.3.7, we have: $LQ_2 \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} LQ_3$ where

$$\begin{aligned} LQ_3 \quad \equiv \quad & \mathbf{new}\ \mathtt{migrated} : \mathtt{\hat{}^{rw}[Site\ \hat{}^{rw}[]]}\ \mathbf{in} \\ & \quad (@_D \mathbf{new}\ m : \mathtt{Map[Agent^s\ Site]}\ \mathbf{in} \\ & \qquad (\mathsf{makeMap}(m; map) \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})) \\ & \quad \mid @_{a_e}(\mathsf{migReady}(s\ \mathtt{migrated}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer})) \\ = \quad & \mathsf{mapS}_{(a_e,\ [Q\ \mathsf{MrdyA}(s\ P)])}(map) \end{aligned}$$

- By transitivity of expansion, $LQ' \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} LQ_3$. By Lemma D.1.3, we have: $a \notin \mathsf{mayMove}(@_D \ldots, \mathcal{F}[\![\boldsymbol{A}']\!])$. Applying Theorem 5.4.3, we have: $LP' \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{E}[LQ_3]$.

Clearly, $\mathcal{E}[LQ_3] \equiv \mathcal{F}[\![Sys']\!]$, thus the lemma is true for this case.

Other cases are given in Appendix D.2 on page 249. ■

Having obtained such operational correspondence results, we may now show that $\mathcal{F}$ is an expansion up to expansion (and therefore an expansion, by Lemma 6.1.1). This involves validating the diagrams below. Note that, since the type system of the intermediate language ensures that all observable agents in a well-formed system are static, translocation of such agents never occur; we therefore omit all translocating indices.

$$\begin{tikzcd} \Phi \Vdash \mathcal{F}[\![Sys]\!] \arrow[r, "\beta", "\Xi"'] \arrow[d, dash, "\mathcal{R}_\Phi"] & \Phi, \Xi \Vdash LP \arrow[d, dotted, dash, "\dot{\succeq}_{\Phi,\Xi}\ \mathcal{R}_{\Phi,\Xi}"] \\ \Phi \Vdash Sys \arrow[r, dotted, "\beta", "\Xi"'] & \Phi, \Xi \Vdash Sys' \end{tikzcd} \qquad \begin{tikzcd} \Phi \Vdash \mathcal{F}[\![Sys]\!] \arrow[r, dotted, "\beta", "\Xi"'] \arrow[d, dash, "\mathcal{R}_\Phi"] & \dot{\succeq}_{\Phi,\Xi}\ \Phi, \Xi \Vdash \mathcal{F}[\![Sys']\!] \arrow[d, dotted, dash, "\mathcal{R}_{\Phi,\Xi}"] \\ \Phi \Vdash Sys \arrow[r, "\beta", "\Xi"'] & \Phi, \Xi \Vdash Sys' \end{tikzcd}$$

Lemma 7.3.5 (Semantic correctness)
For any Sys, with $\Phi \vdash Sys\ ok$, $\mathcal{F}[\![Sys]\!] \dot{\succeq}^{\emptyset}_{\Phi} Sys$.

Proof: Construct a translocating indexed relation $\mathcal{R}$ by

$$\mathcal{R}_\Phi \stackrel{\text{def}}{=} \{(\mathcal{F}[\![Sys]\!], Sys) \mid \Phi \vdash Sys\ ok\ \}.$$

We then use Lemmas 7.3.3 and 7.3.4 for proving that $\mathcal{R}$ is an expansion up to expansion. Explicit name substitution and alpha-conversion are used for ensuring that the names in Φ_{aux} are never extruded by an input or output action by Sys or $\mathcal{F}[\![Sys]\!]$. Once the claim is established, we apply Lemma 6.1.1 to obtain the lemma in its exact form. ■

7.4 Decoding Systems into Source Programs

This section turns to the relationship between the intermediate language and the source language. Since systems in the intermediate language may be in partially committed states, we define *undo* and *commit* decoding functions, $\mathcal{D}^\flat$ and $\mathcal{D}^\sharp$, for mapping systems to source-level terms. We repeat a diagram given at the beginning of this chapter below.

$$\mathrm{n}\pi_{\mathsf{LD,LI}} \underset{\mathcal{D}^\sharp[\![\cdot]\!]}{\overset{\mathcal{D}^\flat[\![\cdot]\!]}{\longleftarrow}} \mathrm{IL}$$

The functions undo and complete partially committed migrations. It suffices to have both functions commit creations and LI messages, as these are confluent. In fact, a create action does not commute with migrate actions by the same agent (since the latter will place the new agent at different site). However, an agent in a partially committed creation state cannot migrate until the creation is fully-committed and subsequently releases the local lock, and so such a situation will never occur.

By being confluent, no processes can distinguish whether such actions have been committed; therefore they can be considered committed once they are initiated (by (SYS-REQ-REG) or (SYS-REQ-MESG)). This is not true for partially committed migrations, for a LD inter-agent communication can easily detect whether an agent has migrated away.

The definitions of $\mathcal{D}$-decoding of pending migration are as follows.

$$\begin{aligned}
\mathcal{D}^\flat[\![\mathsf{MtingA}(s\ P)]\!]_a &\stackrel{\text{def}}{=} @_a\mathbf{migrate\ to}\ s \to P\\
\mathcal{D}^\flat[\![\mathsf{MrdyA}(s\ P)]\!]_a &\stackrel{\text{def}}{=} @_a\mathbf{migrate\ to}\ s \to P\\
\\
\mathcal{D}^\sharp[\![\mathsf{MtingA}(s\ P)]\!]_a &\stackrel{\text{def}}{=} @_a P\\
\mathcal{D}^\sharp[\![\mathsf{MrdyA}(s\ P)]\!]_a &\stackrel{\text{def}}{=} @_a P
\end{aligned}$$

Committing a pending migration of an agent a has the effect of updating the site annotation of a in the top-level binding (we recall that mobile agents are always bound in well-formed systems). The definitions of the undo and commit decodings on systems are therefore slightly different.

$$\begin{aligned}
\mathcal{D}^\flat[\![\mathsf{eProg}(\Delta;\boldsymbol{D};\boldsymbol{A})]\!] &\stackrel{\text{def}}{=} \mathbf{new}\ \Delta\ \mathbf{in}\ \mathcal{D}^\flat[\![\boldsymbol{D}]\!] \mid \mathcal{D}^\flat[\![\boldsymbol{A}]\!]\\
\mathcal{D}^\sharp[\![\mathsf{eProg}(\Delta;\boldsymbol{D};\boldsymbol{A})]\!] &\stackrel{\text{def}}{=} \mathbf{new}\ \Delta\mathsf{commEff}(\boldsymbol{A})\ \mathbf{in}\ \mathcal{D}^\sharp[\![\boldsymbol{D}]\!] \mid \mathcal{D}^\sharp[\![\boldsymbol{A}]\!]
\end{aligned}$$

The effect of committing pending migrations in the state $\boldsymbol{A}$ of agent, written $\mathsf{commEff}(\boldsymbol{A})$, is a relocator defined as follows.

$$\mathsf{commEff}(\boldsymbol{A}) \stackrel{\text{def}}{=} \mathsf{commEff}_{a_1}(\boldsymbol{A}(a_1)) \ldots \mathsf{commEff}_{a_n}(\boldsymbol{A}(a_n))$$
$$\mathsf{dom}(\boldsymbol{A}) = \{a_1, \ldots, a_n\}$$
$$\mathsf{commEff}_a([P\ \boldsymbol{E}]) \stackrel{\text{def}}{=} \begin{cases} a \mapsto s & \boldsymbol{E} = \mathsf{MtingA}(s\ P) \\ a \mapsto s & \boldsymbol{E} = \mathsf{MrdyA}(s\ P) \\ \bullet & \text{otherwise} \end{cases}$$

The rest of the definition, which concerns non-partial processes, is the same for both decoding functions (read $\mathcal{D}$ as either $\mathcal{D}^\flat$ or $\mathcal{D}^\sharp$).

$$\begin{array}{lcl} \mathcal{D}\,[\![[map\ \mathsf{mesgQ}]]\!] & \stackrel{\text{def}}{=} & \mathcal{D}\,[\![\mathsf{mesgQ}]\!] \\ \mathcal{D}\,[\![\prod_{i\in I} \mathsf{mesgReq}(\{T_i\}\ [a_i\ c_i\ v_i])]\!] & \stackrel{\text{def}}{=} & \prod_{i\in I} @_{a_i} c_i \,\boldsymbol{!}\, v_i \\ & & \\ \mathcal{D}\,[\![\boldsymbol{A}]\!] & \stackrel{\text{def}}{=} & \prod_{a\in\mathsf{dom}(\boldsymbol{A})} \mathcal{D}\,[\![\boldsymbol{A}(a)]\!]_a \\ \mathcal{D}\,[\![[P\ \boldsymbol{E}]]\!]_a & \stackrel{\text{def}}{=} & @_a(P) \mid \mathcal{D}\,[\![\boldsymbol{E}]\!]_a \\ & & \\ \mathcal{D}\,[\![\mathsf{FreeA}(s)]\!]_a & \stackrel{\text{def}}{=} & @_a\mathbf{0} \\ \mathcal{D}\,[\![\mathsf{RegA}(b\ Z\ s\ P\ Q)]\!]_a & \stackrel{\text{def}}{=} & \mathtt{new}\ b : \mathtt{Agent}^Z @s\ \mathtt{in}\ (@_b P \mid @_a Q) \end{array}$$

Decoding the state of the daemon commits all the message forwarding requests; the site map is ignored. Decoding the state of agents combines all decoded agents by parallel composition. Decoding the state of an agent places the main body and the undone/committed pending process in parallel. Note that although a pending creation is always committed, the daemon has not yet updated its site map, and therefore the new agent binding is not placed in the top-level declaration.

The following lemma makes sure that decoding a newly-loaded process LP results in LP itself. This is essential for Theorem 7.5.1.

Lemma 7.4.1 (Decoding loaded system)
If $\vdash \Phi\ ok$ and $\Phi \vdash LP$ then $\mathcal{D}^\flat\,[\![\mathcal{L}_\Phi\,[\![LP]\!]]\!] \equiv \mathcal{D}^\sharp\,[\![\mathcal{L}_\Phi\,[\![LP]\!]]\!] \equiv LP$.

Proof: Assume, without lost of generality, that

$$LP \equiv \mathtt{new}\ \Delta\ \mathtt{in}\ (@_{a_1}P_1 \mid \ldots \mid @_{a_n}P_n)$$

where $a_1 \ldots a_n$ are distinct; moreover, $\Phi \vdash a_i@s_i$ for all i. By definition, we have:

$$\mathcal{L}_\Phi\,[\![LP]\!] \;=\; \mathsf{eProg}(\Delta; [\mathsf{Enlist}(\Phi,\Delta)\ \bullet]; \boldsymbol{A})$$

where $\boldsymbol{A}$ is defined so that $\boldsymbol{A}(a_i) = [P_i \ \mathsf{FreeA}(s_i)]$.

This means that the daemon is idle, and

$$
\begin{aligned}
\mathcal{D}\llbracket \mathcal{L}_\Phi \llbracket LP \rrbracket \rrbracket &= \mathbf{new}\ \Delta\ \mathbf{in}\ \mathcal{D}\llbracket [\mathsf{Enlist}(\Phi,\Delta)\ \bullet] \rrbracket \ \mid \mathcal{D}\llbracket \boldsymbol{A} \rrbracket \\
&= \mathbf{new}\ \Delta\ \mathbf{in}\ \mathcal{D}\llbracket \bullet \rrbracket \ \mid \prod_{i\in 1\ldots n} \mathcal{D}\llbracket [P_i \ \mathsf{FreeA}(s_i)] \rrbracket_{a_i} \\
&= \mathbf{new}\ \Delta\ \mathbf{in} \prod_{i\in 1\ldots n} @_{a_i} P_i
\end{aligned}
$$

■

7.4.1 Behavioural Properties

The following two lemmas give operational correspondences between terms in the intermediate language and their decoded images. We prove that a system is strictly simulated by its undo-decoded image, and that a system progressing simulates its commit-decoded image. The results can be summarised in the diagrams below.

$$
\begin{tikzcd}
\Phi \Vdash Sys \arrow[r, "\beta", "\Xi"'] \arrow[d, dash, "\mathcal{R}_\Phi"'] & \Phi,\Xi \Vdash Sys' \arrow[d, dash, dotted, "\mathcal{R}_{\Phi,\Xi}"] \\
\Phi \Vdash \mathcal{D}^\flat \llbracket Sys \rrbracket \arrow[r, dotted, "\hat{\beta}", "\Xi"'] & \Phi,\Xi \Vdash \mathcal{D}^\flat \llbracket Sys' \rrbracket
\end{tikzcd}
\qquad
\begin{tikzcd}
\Phi \Vdash \mathcal{D}^\sharp \llbracket Sys \rrbracket \arrow[r, "\beta", "\Xi"'] \arrow[d, dash, "\mathcal{R}_\Phi"'] & \Phi,\Xi \Vdash LP \arrow[d, dash, dotted, "\equiv\mathcal{R}_{\Phi,\Xi}"] \\
\Phi \Vdash Sys \arrow[r, Rightarrow, dashed, "\beta", "\Xi"'] & \Phi,\Xi \Vdash Sys'
\end{tikzcd}
$$

Lemma 7.4.2 ($\mathcal{D}^\flat$ is a strict simulation)
For any Sys with $\Phi \vdash Sys\ ok$, if $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ then $\Phi \Vdash \mathcal{D}^\flat \llbracket Sys \rrbracket \xrightarrow[\Xi]{\hat{\beta}} \mathcal{D}^\flat \llbracket Sys' \rrbracket$.

Lemma 7.4.3 ($\mathcal{D}^{\sharp^{-1}}$ is a progressing simulation)
For any Sys, with $\Phi \vdash Sys\ ok$, if $\Phi \Vdash \mathcal{D}^\sharp \llbracket Sys \rrbracket \xrightarrow[\Xi]{\beta} LP$ then there exists Sys' such that $LP \equiv \mathcal{D}^\sharp \llbracket Sys' \rrbracket$ and $\Phi \Vdash Sys \overset{\beta}{\underset{\Xi}{\Longrightarrow}} Sys'$.

The details of the proofs of these lemmas are given in Appendix D.3 on page 265; we shall outline them here. The proof of Lemma 7.4.2 is relatively straightforward, using an induction on transition derivation. To prove Lemma 7.4.3, however, it is more convenient to analyse transitions of *fully-committed* system, where all local locks and the daemon lock are free. In a fully committed system, whenever its commit-decoding image perform an action, the system can immediately performs the same action (or initiate it, if the action is daemon-dependent), since an agent does not have to wait for the completion of a pending process. For each system Sys, well-typed w.r.t. Φ, there exists a corresponding fully-committed state, denoted $\mathsf{Norm}_\Phi(Sys)$, defined below.

$$\begin{array}{lll}
\mathsf{Norm}_\Phi(\mathsf{eProg}(\Delta;\boldsymbol{D};\boldsymbol{A})) & \stackrel{\mathrm{def}}{=} & \mathsf{eProg}(\Delta;\boldsymbol{D};\boldsymbol{A}) \\
& & \quad \text{if } \forall\, a \in \mathsf{dom}(\boldsymbol{A}) \;.\; \exists P, s \;.\; \boldsymbol{A}(a) = [P\ \mathsf{FreeA}(s)] \\
\mathsf{Norm}_\Phi(\mathsf{eProg}(\Delta;[map\ \mathsf{mesgQ}];\boldsymbol{A})) & \stackrel{\mathrm{def}}{=} &
\end{array}$$

$$\begin{cases}
\mathsf{Norm}_\Phi\Big(\mathsf{eProg}\Big(\begin{array}{l}\Delta,\ b : \mathtt{Agent}^Z@s;\ [[b\ s]\mathbf{::}map\ \mathsf{mesgQ}]; \\ \quad \boldsymbol{A} \oplus a \mapsto [P \mid R\ \mathsf{FreeA}(s)] \oplus b \mapsto [Q\ \mathsf{FreeA}(s)]\end{array}\Big)\Big) \\
\qquad\qquad\qquad \text{if } \boldsymbol{A}(a) = [P\ \mathsf{RegA}(b\ Z\ s\ Q\ R)] \text{ and } b \notin \mathsf{dom}(\Phi,\Delta) \\[2ex]
\mathsf{Norm}_\Phi(\mathsf{eProg}(\Delta;[[a\ s]\mathbf{::}map\ \mathsf{mesgQ}];\boldsymbol{A} \oplus a \mapsto [P\ \mathsf{FreeA}(s)])) \\
\qquad\qquad\qquad \text{if } \boldsymbol{A}(a) = [P\ \mathsf{MtingA}(s\ Q)] \\[2ex]
\mathsf{Norm}_\Phi(\mathsf{eProg}(\Delta;[[a\ s]\mathbf{::}map\ \mathsf{mesgQ}];\boldsymbol{A} \oplus a \mapsto [P\ \mathsf{FreeA}(s)])) \\
\qquad\qquad\qquad \text{if } \boldsymbol{A}(a) = [P\ \mathsf{MrdyA}(s\ Q)]
\end{cases}$$

Full-commitment preserves the committing decoding. Moreover, any system can always reach a fully-committed state by a sequence of silent actions.

Lemma 7.4.4 (Fully-committed system)
For any Sys with $\Phi \vdash Sys\ ok$, we have:

1. $\mathcal{D}^\sharp [\![Sys]\!] \equiv \mathcal{D}^\sharp [\![\mathsf{Norm}_\Phi(Sys)]\!]$; and
2. $\Phi \Vdash Sys \Longrightarrow \mathsf{Norm}_\Phi(Sys)$.

Proof: First we define two functions, $\mathsf{agentN}(\cdot)$ and $\mathsf{uncommitN}(\cdot)$: they give the number of all agents in the system, and of agents which are not in a partially committed state. The formal definition is given below.

$$\begin{array}{rcl}
\mathsf{agentN}(\mathsf{eProg}(\Delta;\boldsymbol{D};\boldsymbol{A})) & = & \mathsf{sizeof}(\boldsymbol{A}) \\
\mathsf{uncommitN}(\mathsf{eProg}(\Delta;\boldsymbol{D};\boldsymbol{A})) & = & \mathsf{sizeof}(\{a \mid \exists s, P \,.\, \boldsymbol{A}(a) = [P\ \mathsf{FreeA}(s)]\})
\end{array}$$

We prove the lemma by an induction on $\mathsf{agentN}(Sys) - \mathsf{uncommitN}(Sys)$. The base case (ie. $\mathsf{agentN}(Sys) = \mathsf{uncommitN}(Sys)$) is trivial, we shall only demonstrate an induction case here.

Assume the lemma is true for any $Sys_k = \mathsf{eProg}(\Delta_k;\boldsymbol{D}_k;\boldsymbol{A}_k)$ with

$$\mathsf{agentN}(Sys_k) - \mathsf{uncommitN}(Sys_k) \;\leq\; k.$$

Suppose Sys, with $\Phi \vdash Sys\ ok$, is such that $\mathsf{agentN}(Sys) - \mathsf{uncommitN}(Sys) = k+1$. Let $Sys = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$ and $a \in \mathsf{dom}(\boldsymbol{A})$. Suppose, without loss of generality, that $\boldsymbol{A}(a) = [P\ \mathsf{MrdyA}(s\ Q)]$. In this case, we have $\mathsf{Norm}_\Phi(Sys) = \mathsf{Norm}_\Phi(Sys')$ where

$$\begin{array}{rcl} Sys' & = & \mathsf{eProg}(\Delta \oplus a \mapsto s; [[a\ s]::map\ \mathsf{mesgQ}]; \boldsymbol{A}') \\ \boldsymbol{A}' & = & \boldsymbol{A} \oplus a \mapsto [P\ \mathsf{FreeA}(s)] \end{array}$$

To obtain the first property, we derive the following.

$$\begin{array}{rcl} \mathcal{D}^\sharp [\![Sys]\!] & = & \mathbf{new}\ \Delta\mathsf{commEff}(\boldsymbol{A})\ \mathbf{in}\ \mathcal{D}^\sharp [\![[map\ \mathsf{mesgQ}]]\!] \mid \mathcal{D}^\sharp [\![\boldsymbol{A}]\!] \\ & = & \mathbf{new}\ (\Delta \oplus a \mapsto s)\mathsf{commEff}(\boldsymbol{A}')\ \mathbf{in}\ \mathcal{D}^\sharp [\![[[a\ s]::map\ \mathsf{mesgQ}]]\!] \mid \mathcal{D}^\sharp [\![\boldsymbol{A}']\!] \\ & = & \mathcal{D}^\sharp [\![Sys']\!] \end{array}$$

Since $\mathcal{D}^\sharp [\![Sys']\!] \equiv \mathcal{D}^\sharp [\![\mathsf{Norm}_\Phi(Sys')]\!]$, by the induction hypothesis. We have $\mathcal{D}^\sharp [\![Sys]\!] \equiv \mathcal{D}^\sharp [\![\mathsf{Norm}_\Phi(Sys)]\!]$. Thus true for this case.

To obtain the second property, we derive the following.

- By (SYS-COMM-MIG), $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$.
- $\Phi \Vdash Sys' \Longrightarrow \mathsf{Norm}_\Phi(Sys')$, by the induction hypothesis.
- $\Phi \Vdash Sys \Longrightarrow \mathsf{Norm}_\Phi(Sys)$, since $\mathsf{Norm}_\Phi(Sys) = \mathsf{Norm}_\Phi(Sys')$.

Thus true for this case. Cases where a is in the state $\mathsf{RegA}(b\ Z\ s\ Q\ R)$ and $\mathsf{MtingA}(s\ Q)$ are similar. ■

We also show that these two decodings form a coupled simulation. To prove this, we need to show that decodings are coupled by validating the following diagrams:

$$\begin{tikzcd} \Phi \Vdash Sys \arrow[d, "\mathcal{D}^\flat"'] \arrow[dr, dotted, "\mathcal{D}^\sharp"] & \\ \Phi \Vdash \mathcal{D}^\flat [\![Sys]\!] \arrow[r, Rightarrow, dashed, "\tau"] & \Phi \Vdash \mathcal{D}^\sharp [\![Sys]\!] \end{tikzcd} \qquad \begin{tikzcd} \Phi \Vdash Sys \arrow[r, Rightarrow, dashed, "\hat{\tau}"] \arrow[d, "\mathcal{D}^\sharp"'] & \Phi \Vdash Sys' \arrow[dl, dotted, "\mathcal{D}^\flat"] \\ \Phi \Vdash \mathcal{D}^\sharp [\![Sys]\!] & \end{tikzcd}$$

In fact the diagram on the right is a direct consequence of Lemma 7.4.4 (i.e. picking $Sys' = \mathsf{Norm}(Sys)$). We are thus only required to validate the diagram on the left.

Lemma 7.4.5 (Coupling of commit and undo decodings)
For any Sys with $\Phi \vdash Sys\ ok$, $\Phi \Vdash \mathcal{D}^\flat [\![Sys]\!] \Longrightarrow \mathcal{D}^\sharp [\![Sys]\!]$.

Proof: If Sys contains k agents which have pending migrations, then $\mathcal{D}^\flat [\![Sys]\!]$ can perform (bound) migration k times to become $\mathcal{D}^\sharp [\![Sys]\!]$. ■

Lemma 7.4.6 (Decodings form a coupled simulation)
$(\mathcal{D}^\flat, \mathcal{D}^\sharp)$ is a coupled simulation, ie. if $\Phi \vdash Sys\ ok$ then $Sys \fallingdotseq_\Phi \mathcal{D}\,[\![Sys]\!]$.

Proof: By Lemma 7.4.2 and Lemma 7.4.3, we know that $\mathcal{D}^\flat$ and $(\mathcal{D}^\sharp)^{-1}$ are weak simulations. By Lemma 7.4.5 and Lemma 7.4.4, we have the necessary coupling between the decoding for $(\mathcal{D}^\flat, \mathcal{D}^\sharp)$ to be a coupled simulation. ∎

7.5 The Main Result

The results from the previous two sections are combined to give a direct relation between the source and the target terms, proving that the source term LP, well-typed w.r.t. valid system context Φ, and its translation $\mathcal{C}_\Phi\,[\![LP]\!]$ are related by a coupled simulation.

Theorem 7.5.1 (Coupled simulation)
For any LP well-formed w.r.t. a valid system context Φ, $LP \fallingdotseq_\Phi \mathcal{C}_\Phi\,[\![LP]\!]$.

Proof: The proof puts together the operational correspondence results developed, as can be summarised in the diagram below.

$$
\begin{tikzcd}
\mathrm{n}\pi_{\mathsf{LD,LI}} & LP \arrow[r, dash, "\equiv", "(7.4.1)"'] \arrow[dd, dotted, dash, "\fallingdotseq_\Phi"', "(5.4.6{,}5.4.5)"] & \mathcal{D}\,[\![\mathcal{L}_\Phi\,[\![LP]\!]]\!] \arrow[d, dash, "\fallingdotseq_\Phi"', "(7.4.6)"] \\
\mathrm{IL} & & \mathcal{L}_\Phi\,[\![LP]\!] \arrow[d, dash, "\dot{\succeq}^{\emptyset}_{\Phi}"', "(7.3.5)"] \\
\mathrm{n}\pi_{\mathsf{LD}} & \mathcal{C}_\Phi\,[\![LP]\!] \arrow[r, dash, "\equiv", "(7.3.2)"'] & \mathcal{F}\,[\![\mathcal{L}_\Phi\,[\![LP]\!]]\!]
\end{tikzcd}
$$

More precisely, we pick $Sys = \mathcal{L}_\Phi\,[\![LP]\!]$ and derive the following.

- $\Phi \vdash Sys\ ok$, by Lemma 7.3.1.
- $\mathcal{D}\,[\![Sys]\!] \equiv LP$, by Lemma 7.4.1.
- $Sys \fallingdotseq_\Phi \mathcal{D}\,[\![Sys]\!]$, by Lemma 7.4.6.
- $\mathcal{F}\,[\![Sys]\!] \dot{\succeq}^{\emptyset}_{\Phi} Sys$, by Lemma 7.3.5.
- $\mathcal{F}\,[\![Sys]\!] \fallingdotseq_\Phi \mathcal{D}\,[\![Sys]\!]$, by Lemma 5.4.6 and Lemma 5.4.5.
- $\mathcal{F}\,[\![Sys]\!] \equiv \mathcal{C}_\Phi\,[\![LP]\!]$, by Lemma 7.3.2.

Hence the theorem. ∎

Chapter 8

Related Models

There exists a vast number of works on verification of distributed and mobile computations. They can be roughly divided into the following.

- **Protocol analysis** This involves formalising an informal description of a *particular* protocol, abstracting away irrelevant aspects, and apply mathematical techniques to discover the system structure and its behavioural properties. The works in this class include analysis of various protocols using the I/O automata, of Mobile IP [AP98, JNW99], of an active network algorithm [WMG00], and the verification of a distributed directory service and message routing algorithm for mobile agents [Mor99].

- **Modelling language** This involves defining a formal language intended for describing a wide range of distributed and mobile computations. The existing modelling languages includes Mobile UNITY and variants of distributed process calculi.

In this chapter, we shall explore some of these works, focusing our attention on those which are:

- equipped with precise semantic definitions;

- flexible, allowing a wide range of distributed and mobile computations to be precisely expressed; and

- equipped with notions of correctness and proof methods, allowing programs to be verified and reasoned about.

With these criteria, we select three models for extensive reviewing: variants of distributed process calculi, the I/O automata and the Mobile UNITY model. The next three sections discuss these models: their basic constructs, their informal semantics, and some of their verification techniques.

There exist many other modelling languages, which are generally based on well-established models of concurrency such as Petri nets [Rei85] and Linda [Gel85]. We omit discussions of models based on Petri nets for they remain somewhat too static to be directly used for mobile computation. Some effort has been made to increase the degree of dynamism, eg. Mobile Petri Nets [AB96], but its only success is to add π-style dynamics of name scoping to the model. Also omitted are discussions of models and languages based on Linda paradigm, such as LLinda [dNFP97b], Bauhaus [CGZ95], Klaim [dNFP97a] and MobilS [Mas99]. These languages are intended as specification languages and come somewhat under-equipped as far as proof and verification techniques are concerned (eg. no behavioural equivalence is given).

The choice of which model to adopt can be difficult to make. Three factors are perhaps most prominent: expressiveness of a model, the level of abstraction required, and the preferred proof methods. Expressiveness is the most important factor when it is clear that certain features are expressible by one model and not the others. For example, since Mobile UNITY can be used for expressing node mobility and disconnected operations, it is more suitable for modelling mobile-device networks than the I/O automata model (which cannot directly express mobility) or process calculi (which cannot directly express transient resource sharing). In most cases, however, the same algorithm or protocol can be expressed in many different models — in such cases the other two factors become more important. Works on protocol analysis are *idealised* in the sense that a verified protocol cannot directly be used for producing an executable version (see also Section 1.3). The process of formalising the informal description of the protocol can abstract away too many details, making the model somewhat less convincing. To one's intuition of program execution, models which use operational semantics (eg. the process calculi) are often closer and less abstract than those which uses axiomatic semantics (eg. the Mobile UNITY model). However, usage of axiomatic semantics often means that the statements of program properties are more concise and easier to prove correct.

Here it is probably worth quoting a passage from Milner's Turing Award lecture [Mil93a]: "I reject the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent computation, which is in a sense the *whole* of our subject—containing sequential computing as well as a well-behaved special area. We need many *levels of explanation*: many different languages, calculi, and theories for the different specialisms."

8.1 Process calculi-based models

The π-calculus (see an introduction in Section 2.1), along with its variations, is a prominent model of concurrency. The attraction comes from its clear treatment of concurrency, the flexible treatment of names in the π-calculus, and the similarity between π asynchronous message passing and reliable datagram communication. Furthermore, process calculi have a well-established semantics, providing a tractable theory and convenient proof techniques for reasoning about process behaviour. Nevertheless many important issues in distributed computing remain unaddressed by the π-calculus. Most crucial of all is the lack of notions of distribution, locality, mobility and security.

To address such issues, many π-based calculi have been proposed. They are often specific, tailored to studied particular problems in distributed and mobile computations. The number of such calculi increases rapidly, so instead of describing each of them in turn, we review some of the common design choices, focusing on the notions of distribution, locality and mobility — omitting any discursive details. This review is based on the design of the following calculi, with brief descriptions summarising the abstracts of the original papers.

- The π_l calculus of Amadio and Prasad [AP94], used for modelling the notions of locality and failure presented in the programming language Facile [TLK96].
- The π_{1l} calculus of Amadio [Ama97], used for studying the distribution of processes to locations, the failures of locations and their detection.
- The Distributed Join Calculus of Fournet et al. [FGL+96], intended for providing a foundation of a programming language for distributed and mobile applications.
- The Seal calculus of Vitek and Castagna [VC98, VC99], intended as a framework for writing secure distributed applications over large scale open networks such as the Internet.
- A distributed CCS of Riely and Hennessy [RH97], used for studying characterisation of barbed congruences for a calculus with locations and failures.
- The Dπ calculus of Riely and Hennessy [RH98, RH99], used for studying *partially-typed* semantics, designed for mobile agents in open distributed systems in which some sites may harbor malicious intentions.
- The D$\pi\lambda$ calculus of Yoshida and Hennessy [YH99a, YH99b], used for studying locality using a calculus of distributed higher-order processes in which not only basic values or channels, but also parameterised processes can be transferred across distinct locations.

- The dpi of Sewell [Sew98], used for studying a type system in which the input and output capabilities of channels may be either global or local.
- The Ambient calculus of Cardelli and Gordon [CG98], a calculus for describing the movement of processes and devices, including movement through administrative domains.
- The $\mathrm{D}\pi_1^{\mathrm{r}}$ of Amadio and Boudol [ABL99], used for showing how a static analysis ensures the receptiveness of channel names, which, together with a simple type system, guarantees a local deadlock-freedom property.
- The Box-π-calculus of Sewell and Vitek [SV99, SV00], used for studying *wrappers*: secure environments that provide fine-grain control of the allowable interaction between them, and between components and other system resources.
- The extension of TyCO with distribution and code mobility [VLS98], which is referred to in this overview as *distributed TyCO*. TyCO (Typed Concurrent Object) [Vas94] is a name-passing process calculus, which allows asynchronous communication between concurrent objects via labelled messages carrying names.
- Nomadic π-calculi, addressed in this thesis.

For other discussion, the reader may refer to "related work" sections of the works cited above, as well as to [Sew00, Car99].

Distribution and Locality Here we are concerned about how processes are grouped and collectively named. The π-calculus does not have any notion of the identity of processes, yet grouping process terms is essentially for describing units of migration, channel synchronisation, sites of failures, and administrative domains (for security). Grouped identity can be anonymous, for example in $\mathrm{D}\pi\lambda$, where distinct groups are separated by $\|$. It is more common, however, to name grouped identities, giving a sense of shared locality. Names of units of grouped processes are often referred to as *locations*, and a process P running at location l is often denoted by $@_l P$ or $l[P]$. The terms *agents* and *sites* are also widely used, although they are preserved for movable units of code and static runtime instances (subject to failures) respectively. The terminology for units of trust (or administrative domains) is less established; the existing terms include *ambient*, *seal* and *box*, adopted by the Ambient, Seal and Box π-calculus respectively. Like channel names, location names are first class values (and hence can be transmitted by communication) and can generally be dynamically created.

UNIQUE NAMING: Location names are not necessary distinct. The Ambient and Box π-calculi allow the existence of two or more locations with the same name. This approach is adopted for studying security in distributed system, where an administrative domain may not have control of the name generation outside its control. Non-unique naming, however, implies that all co-located processes must be grouped together so that groups sharing the same name can be distinguished. This style of "united-grouping" can make operational semantics complicated (see eg. [CV99, GC99]), since the structural congruence rule $l[P] \mid l[Q] \equiv l[P \mid Q]$ is excluded. Such a rule allows location information to be projected to atomic processes, and is undesirable for reasoning about security domains, since $l[P]$ may come from an insecure source and must not be mixed with the secured $l[Q]$. For simplicity, unique naming is adopted by most calculi, and is usually enforced by the design of the calculus syntax or typing conditions.

Structure of Locations: Locations can be given a flat structure — a distributed system then simply consists of a set of locations. This approach is adopted by the π_l, distributed CCS, D$\pi\lambda$, Dπ_1^{r} and distributed TyCO. To support migrations (see later), locations need to be organised into some form of hierarchy: Nomadic π supports a two-level hierarchy, whereas the Dπ, Ambient, Distributed Join, dpi, Seal and Box π-calculi support arbitrary tree-structured hierarchy. When a new location is created in a calculus with arbitrary tree-structured locations, the new location becomes a sublocation of the creating location. The tree of locations may be deduced from the process term syntax, as is the case for the Ambient, Seal and Box π-calculi; for example, the term $n[P| \, n'[Q]] \mid m[R]$ induces a location tree with m, n as children of a root node and n' as that of n. In some calculi, the location tree is not maintained in the process syntax, but, for example, in the operational semantics for the Distributed Join, and in the binding located type context for the Nomadic π-calculi.

Mobility The π-calculus is a calculus of mobile processes only as far as scopes of names are concerned; π-processes never actually move. With a notion of distribution, mobility of processes (in the sense of moving an executing process from one location to another) can be modelled precisely.

There are two common semantics of process mobility in distributed calculi, closely corresponding to the mobile code and mobile agent paradigms of mobile code languages (cf. Section 1.1).

1. **Spawning** In calculi with flat location structure, a spawning primitive eg. $\texttt{spawn}(l, P)$ allows the process P to be executed at location l. This closely corresponds to the mobile code paradigm, where only *code* move between nodes. Note that, in addition to spawning, distributed TyCO also allows definition of template processes **def** D to be downloaded from one site to another, where a template process is similar to a

parametric process definition and join patterns (see more in "variants of communication primitives" in this section).

2. **Migration** With hierarchical locations, process mobility can be modelled by changes of location-tree structure. Migration closely corresponds to the mobile agent paradigm, where *states of execution* as well as codes can move between nodes. As an example, in case of Nomadic π-calculi (with two-level hierarchy of agents located on sites) an agent a may migrate between sites, moving all the processes located at a with it. More generally, the calculi with an arbitrary location-tree structure offer *structured migration*, where executing a migration primitive has an effect of moving a location and all its descendents to be a sublocation of another location. This style of migration is similar to the *fine-grained mobility* supported by the programming language Emerald [JLHB88].

There exists other paradigms of process mobility in distributed computing, an example of which is that of Obliq [Car95], where objects are never moved, but rather *cloned* to different sites. It is claimed that the migration, if necessary, can be implemented by means of cloning and aliasing. Sekiguchi and Yonezawa formulated λdist [SY97] for studying various code moving mechanisms, giving them precise semantics. We omit further discussion here.

Interaction Channel communication remains the main means of computation in distributed process calculi. With a precise notion of distribution, it is now possible to describe how channel communication across location boundaries occurs. We distinguish two major semantics of interaction.

1. **Location-transparent interaction** In this semantics, locations play no role in channel communication (provided they are not failed) — the details of message routing are hidden from the programmers. Location-transparent interaction can be *non-local*, ie. an input and an output on the same channel may interact regardless of where they are located. For example, the output $x!v_1$ at a_1 in the process

$$@_{a_1}x!v_1 \mid @_{a_2}x!v_2 \mid @_{a_3}x?p \rightarrow P \mid @_{a_4}x?p \rightarrow Q$$

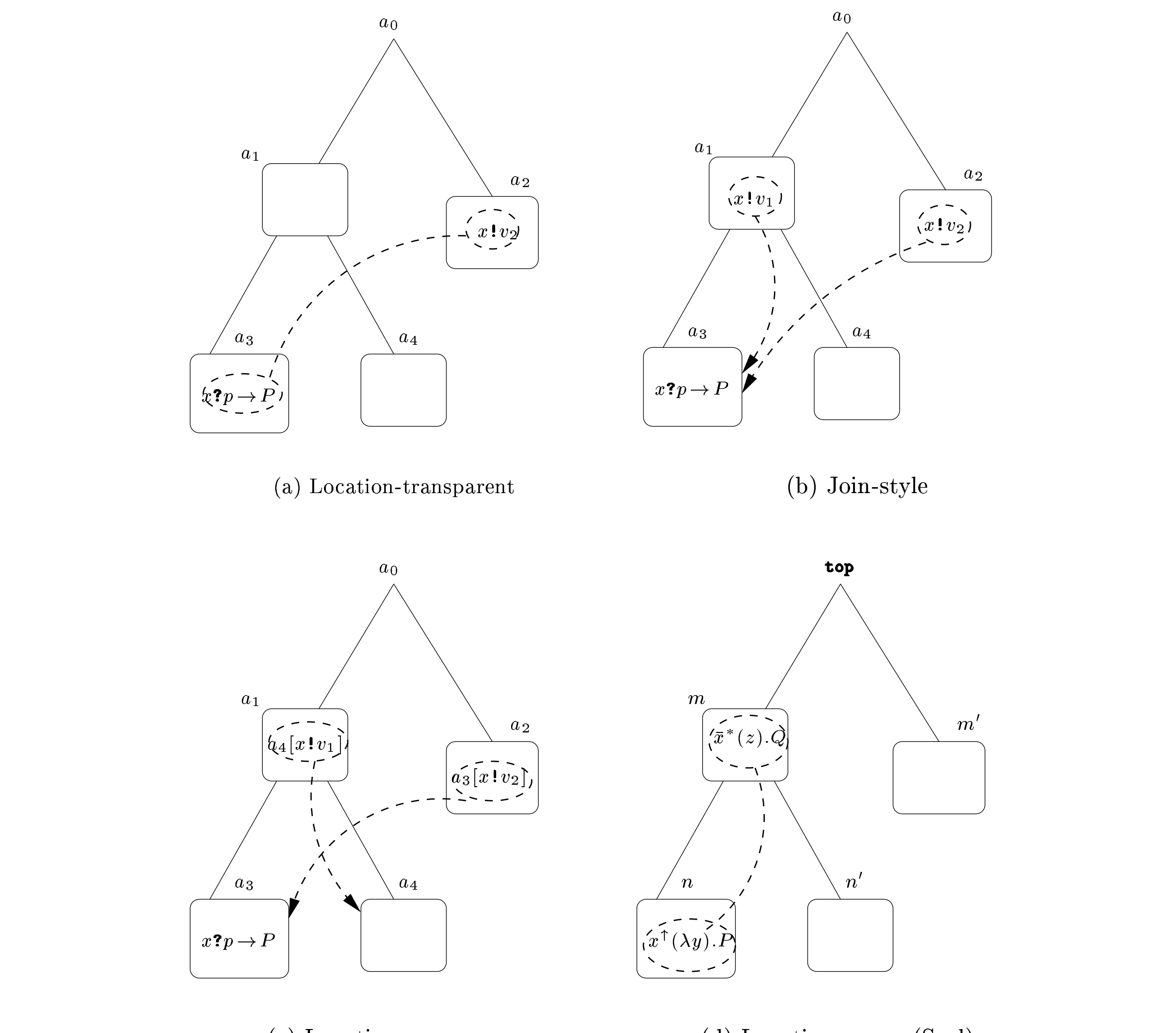
may interact with an input on x at a_3 or a_4; the same also holds for the output $x!v_2$ at a_2. Figure 8.1(a) shows an interaction of a pair of input and output on x at locations a_2 and a_3. This style is adopted by the π_l, dpi, and other calculi whose group identity is anonymous.

Fournet and Gonthier [FG96] argued that non-local interaction is difficult to implement efficiently in a distributed setting. They formulated the Distributed Join calculus, based on a chemical abstract machine [Bou94]. The calculus retains the location-transparent interaction semantics since programmers need not specify where channel interaction may take place. However, efficiency is gained by the restriction that each channel is *defined* in at a unique location and only at its "defining" location can values be read from a channel — outputs on such a channel have to travel to the defining location in order to interact with an input. Figure 8.1(b) shows messages on x moving from the locations a_1 and a_2 to the defining location of x, a_3, where an input resides. As there exists a unique defining location of each channel, transport is deterministic and point-to-point — there is no need for handshaking before communicating on channels.

We shall refer to the latter style of interaction as *join-style* interaction. It has a great influence on the design of many subsequent calculi, such as $\mathrm{D}\pi_1^{\mathrm{r}}$, $\mathrm{D}\pi\lambda$, dpi, Ambient, as well as Nomadic π-calculi.

2. **Location-aware interaction** Another way of making channel interaction efficiently implementable in a distributed setting is by making it location-aware. In this case, an input and an output on the same channel may synchronise only if they are co-located, or, in the case of the Seal and Box-π calculi, if they are in close vicinity (eg. on the same physical machine). Interaction between arbitrary locations is also possible. In most calculi, the programmers are required to *explicitly* specify the location where an output is to be moved to in order to synchronise with an input on the same channel. For example, $@_{a_1}(@_{a_4}x\,\mathbf{!}\,v_1) \mid @_{a_2}(@_{a_3}x\,\mathbf{!}\,v_2) \mid @_{a_3}x\,\mathbf{?}p \rightarrow P$ reduces to $@_{a_1}(@_{a_4}x\,\mathbf{!}\,v_1) \mid @_{a_3}\mathsf{match}(p, v_2)P$, and never $@_{a_2}(@_{a_3}x\,\mathbf{!}\,v_2) \mid @_{a_3}\mathsf{match}(p, v_1)P$. Figure 8.1(c) illustrates the movement of messages occurs in the example process. This semantics is adopted by the $\mathrm{D}\pi$, $\mathrm{D}\pi_1^{\mathrm{r}}$, distributed TyCO, and the high-level Nomadic π-calculus.

   In a calculus with tree-structured locations, an agent a may communicate with another agent b if a correctly specifies an explicit path from a to b, or a relative location in the tree which is the parent of b. For the sake of later discussion, we shall refer to this style of interaction as being *location-path specific*. We have two examples of this. In low-level Nomadic π, an agent a may send a message to another agent b if the two agents are at the same site; the path in this case is from a to its parent (ie. its site) and then to b. The (sugared) primitive for inter-agent interaction, $\langle b@s\rangle c\,\mathbf{!}\,v$, specifies the relative location s which is the parent of b. In the Seal and Box π-calculi communication between seals (or boxes) may occur only if they are parent and child. Process expressions of the Seal calculus include $x^n(\lambda y).P$, which tries to read a value along channel x located in the child named n, and $\bar{x}^{\uparrow}(y).P$, which tries to output a value along channel x located in the parent. Local inputs and outputs are denoted by $x^*(\lambda y).P$ and $\bar{x}^*(y).P$ respectively. For example, an expression $n[x^\uparrow(\lambda y).P] \mid \bar{x}^*(z).Q$ reduces to $n[\{z/y\}\ P] \mid Q$, as illustrated in Figure 8.1(d). Seal-style primitives keep tight control over channel names, allowing local resources to be protected from malicious components.



(a) Location-transparent (b) Join-style

(c) Location-aware (d) Location-aware (Seal)

Figure 8.1: Channel communication in distributed setting

Location dependency: We may classify the described interaction styles as being location dependent or location independent. Location-transparent interaction is definitely location-independent, since it allows agents to interact irrespective of their locations. Location-path-specific interaction can be classified as location-dependent. If a calculus allows locations to be structured as a tree, a location-aware interaction which is *not* location-path specific can be classified as location-independent; this is the case for the high-level primitive of Nomadic π-calculi.

It is hard to classify location-aware interaction in a calculus with flat-structured locations, since locations remain static and there is no need to track their movement. If locations cannot be dynamically created by programs, we may consider them as physical sites. In this case, the interaction style is location-dependent, and is supported by the standard network technology. In most calculi, however, locations can be dynamically created by programs. The fact that these locations actually reside in some physical machines must be abstracted away, which means that some implicit mechanism is required for associating such locations to physical machines. In this case, the interaction style can therefore be classified as location-dependent.

VARIANTS OF INTERACTION PRIMITIVES: Some of the cited calculi offer variants of communication constructs. The join calculus [FG96] replaces inputs and replicated inputs by *join patterns*, allowing many messages (from many sending processes) to synchronise in one action. For example, a join pattern

$$\textbf{def}\ \texttt{count}\langle n\rangle|\texttt{inc}\langle\rangle \triangleright \texttt{count}\langle n+1\rangle\ \textbf{in}\ P$$

allows messages $\texttt{count}\langle 5\rangle$ and $\texttt{inc}\langle\rangle$ in P to simultaneously interact to produce $\texttt{count}\langle 6\rangle$; the names `count` and `inc` are bound in P, preventing them from being input elsewhere. A join pattern can therefore be thought of as a combination of π restriction and replicated input, which also enforces linearity of the channels involved. In TyCO, asynchronous messages and inputs are replaced by the following.

$x\,\mathbf{!}\,l[\vec{v}]$	message
$x\mathbf{?}\{l_1(\vec{x}_1) = P_1 \ldots l_n(\vec{x}_n) = P_n\}$	object

where $\vec{v}, \vec{x}_1, \ldots, \vec{x}_n$ are sequences of names and $l, l_1, \ldots, l_n$ are labels. A message $x\,\mathbf{!}\,l_i[\vec{v}]$ selects the method l_i in an object $x\mathbf{?}\{l_1(\vec{x}_1) = P_1 \ldots l_n(\vec{x}_n) = P_n\}$; the result is the process P_i where names in $\vec{v}$ replace those of $\vec{x}_i$. In comparison to a standard π-calculus, TyCO processes are more like objects in the sense that communication are similar to method invocation, consisting of a name tagged with a label. The expressive power of these variants are addressed by Nestmann in [Nes98] (for join patterns), and, in collaboration with Ravara, [NR00] (for TyCO objects).

Another style of interaction is offered by the Ambient calculus, where computation is based solely on movement of ambients. Instead of sending or receiving along a channel, a process in an ambient m is capable of three basic operations (called *capabilities*). The execution of *in* n in an ambient m moves m into an adjacent ambient n (and becomes its descendant).

The execution *out* n in an ambient m moves m out of its immediate enclosing parent n (and becomes its sibling). The boundary of an ambient n can be dissolved by its immediate enclosing ambient m (thereby "collapsing" a node in the location tree) by executing a primitive *open* n in m. These operations would block if the relative location of the executing ambient does not satisfy that required by the capabilities: for example, for an ambient m to perform *in* n capability, m and n must be siblings. These three basic capabilities, along with π style restriction, parallel composition and replication prove to be sufficiently expressive to express Church's numerals, Turing machines and an encoding of the π-calculus [CG98]. The calculus is relatively simple, yet can be used for reasoning about mobility and security.

Failures In many cases a notion of locations alone may not be sufficient, since locations could all really be in the "same place". However, in presence of failures one could observe that certain locations have failed and others have not, and deduce that those locations are truly in different places; otherwise they would all have failed at the same time. The Distributed Join calculus, for example, adds a notion of location failure, where if a location fails then all processes located at such a location or its descendents are no longer active. Modelling failures is beyond the scope of this thesis, and we omit further discussion.

Security Security in the Internet and the World-Wide-Web has been a subject of increasing concern. We have discussed the primitives of the Ambient, Seal and Box π-calculi, which allow interaction across administrative domains to be controlled. Alternatively, the spi calculus of Abadi and Gordon [AG99] extends the π-calculus with cryptographic primitives, allowing security protocols for authentication and electronic commerce to be described and analysed. The calculus is not distributed, though. The problem of implementing authentication in distributed systems has been addressed by Abadi, Fournet and Gonthier [AFG00], extending the Distributed Join calculus with some cryptographic primitives. Recently, Abadi and Fournet introduced a simple, general extension of the π-calculus with value passing (not merely names), primitive functions, and equations among terms [AF01]; they also showed how security protocols can be expressed in this calculus. Again network security is beyond the scope of this thesis, and we omit further discussion.

8.1.1 Verification and Proof Techniques

Roughly speaking, the verification and proof techniques of process calculi can be classified as those based on types and those based on the dynamic behaviour of processes. A type system for the π-calculus was first proposed by Milner [Mil93b], giving the notions of *sort* and *sortings*, which ensure uniformity of the kind of names that can be sent or receive by channels. Many refinements on the type system have subsequently been proposed, including polymorphism [Tur96, PS97], subtyping [PS96], linear types [KPT96], objects [Wal95], and a generic type system [IK01]. Adding the notions of locality and distribution to the π-calculus admits further refinements to be made. Sewell [Sew98] formulated *dpi* for studying a type system where each channel is located at an agent and can be given global/local usage capability as well as that for input/output. An approximation to the join-style of interaction, for example, can be obtained by giving them global-output and local-input capabilities. This type system retains the expressiveness of channel communication, yet admits optimisation at compile time. Yoshida and Hennessy [YH99b] formulated a type system for $\mathrm{D}\pi\lambda$ which emulates the join-style of interaction using input/output subtyping. The presence of higher-order processes makes this formulation challenging. The type system of $\mathrm{D}\pi_1^{\mathrm{r}}$ extends the concept of uniform receptiveness [San99] to ensure that each output (perhaps an inter-agent message) is guaranteed to react with a (unique) input process at its destination (cf. Section 6.2.3). The techniques of refining of channel types are also used in ensuring security-related properties. For example, the partial typing of Riely and Hennessy [RH99] ensures that resources of trusted sites are not abused by untrusted sites; Sewell and Vitek introduced causality types [SV00] for reasoning about information flow between security domains; and Cardelli, Ghelli and Gordon [CGG00] introduced a notion of *groups* which can be used for ensuring that the boundary of an ambient may only be dissolved by trusted groups of ambients. It is essential the properties ensured by a type system remain valid during execution of a well-typed program. The proof of subject reduction is therefore an essential part of the cited works.

The behavioural theories of these distributed variants of process calculi are generally adapted from those of the π-calculus, which are based around operational semantics and operational equivalences (see Section 5.1 for background). A reduction semantics is given for all of the quoted calculi. This, together with some notions of barbs, allows a definition of barbed bisimulation to be given, as is the case for the Distributed Join-calculus [FG96], the Seal calculus [CV99], and the Ambient calculus [GC99]. A labelled transition semantics is also given for the π_{1l}, $\mathrm{D}\pi$, $\mathrm{D}\pi_1^{\mathrm{r}}$, Ambient, Seal and Box π-calculi, allowing some notions of bisimilarity to be given. These definitions of labelled transition semantics often involve refining that of the standard π with location annotation ($@_l$ for $\mathrm{D}\pi_{\mathrm{r}}^1$ and "relative location" tags for Seal and Box-π). The labelled transition semantics of $\mathrm{D}\pi$ [RH98] extends the standard π input and output actions with labels that indicate movements and failures of locations. The style of LTS of the Ambient calculus [GC99] is quite different from that of π-calculus for it involves relative locations of ambients. The definitions of labelled transition semantics and operational equivalences of distributed CCS [RH97] and π_{1l} [Ama97] also take location failures into account.

Modal logic [HM85] is also useful in reasoning about temporal properties of processes. Logical rules can be used for showing that a process P satisfies a logical formula A, written $P \vDash A$, where each logical formula can be constructed from sequences of actions, equality of names, and boolean connectives $\wedge$ and $\neg$. This allows various forms of behavioural properties of processes to be stated, such as P always receives from a channel c before sending on d. Milner et al. [MPW93] formulated various modal logics for the π-calculus, and showed that two of them can be used for characterising the early and the late bisimulation equivalence. Application of modal logic to distributed process calculi was pioneered by Cardelli and Gordon, who presented a modal logic for the Ambient calculus in [CG00] and later extended it to include name restriction [CG01]. Their logic allows statement and verification of *spatial-modality* properties which hold at certain locations, at some locations, or at every location. Examples of such properties include "eventually the agent crosses the firewall," "somewhere there is a virus" and in their latter work, "a shared key is established between locations a and b."

There is also some work on mechanising verification of reactive systems' properties, specified by some process calculus. The Concurrency Workbench [CPS93, CS96, MS] and the Jack (Just Another Concurrency Kit) environment [BGL94] are automated tools for verifying behavioural and (modal) logical properties of finite-state CCS-like processes (which have no name-passing ability). The Mobility Workbench [VM94, Vic94] is a tool designed specifically for the π-calculus. It can be used for deciding the open π bisimulation equivalences [San96b]. The works described are tools which are specifically designed for process calculi; there are also a number of works on embedding process calculi in existing theorem provers. Hirschkoff [Hir97, Hir98] develops an implementation of the π-calculus in the Coq system [DFH$^+$93], allowing many "classical" results of the π-calculus theory to be proved using the up to context technique [San95b]. Nesi [Nes92] used the HOL system [Gor87] for defining CCS and its modal logic. This was a starting point for the work of Aït-Mohamed, which consisted in proving equivalences and building a system to prove bisimilarities between π-processes interactively [Aït95].

8.1.2 Programming Languages

The design of many concurrent and distributed programming languages has been influenced by the π-calculus. Pierce and Turner [PT00] used an asynchronous choice-free π-calculus as the basis for the programming language Pict (cf. Section 2.1.2 on page 16). The language is not distributed, though. Three of the distributed process calculi described in this section have been used as the basis for a truly distributed programming language: the Join language [FM97], based on the Join calculus, DiTyCO [LSFV99, LAFSV00], based on distributed TyCO, and Nomadic Pict (cf. Section 1.4 on page 7). The implementation of the Join language was developed in the Objective-Caml environment [LDG$^+$00]. To its base calculus, the language adds basic types and most Objective-Caml libraries as primitives, and gives a simple interface to incorporate any other Objective-Caml module. A Join language program can be distributed among on many heterogenous machines, and can contain mobile agents. The language also supports an abstract model of failure and failure detection, although not all runtime failures can be detected. According to [LAFSV00], the first DiTyCO prototype is in the final stages of implementation; the authors also mentioned adding some mechanism for failure detection in the future.

There are also some programming languages which combine concepts from functional languages and the π calculus. Two prominent examples of such languages are Facile [TLP$^+$93] and CML [Rep92], which add concurrency primitives and channel communication to Standard ML [MTH90]. The semantics of the (non-distributed) Facile was explored by Amadio et al. [Ama94, ALT95]; they gave some notion of program equivalence and a translation of Facile into the π-calculus. The distributed programming features of Facile [TLG92] provides transparent access to distributed resources; its semantics was studied via π_l calculus [AP94], also discussed in this section. The reduction semantics of CML was originally proposed by Reppy [Rep92] and Berry et al. [BMT92]. This semantics does not support a notion of program equivalence, though. Two fragments of CML, μCML [FHJ98] and $\mu\nu$CML [JR00], are given a labelled transition semantics and some notions of bisimulation equivalence, allowing program rewrites (eg. in compiler optimizations) to be reasoned about. These semantics of CML are non-distributed, although Jeffrey and Rathke [JR00] claimed that thread identifiers of CML are similar to locations in distributed process calculi and that their treatment of thread could be adapted to distributed setting.

JoCaml [Fes98] also provides a way of combine the benefits of functional programming with concurrency, distribution and mobility. It is an extension of Objective Caml [LDG$^+$00], an implementation of ML, with the Join-calculus constructs (concurrency, synchronization, distribution and mobiles agents) provided as a library. This language provides total transparency for migration, join-style interaction, mobile agents and failure detection. It has been used in several applications, including a mobile editor, some distributed games and a distributed implementation of Ambient calculus.

The design of the programming language Oz [Smo95b] (also known as Mozart [CP99] in its latest release) is also somewhat related to process calculi. Oz offers *multiparadigm* programming, designed to support many different programming paradigms (functional, logic, object-oriented, sequential and concurrent programming); readers may refer to [MMvR95] for more details. Its underlying formalism, the actor model [Smo95a], is an extension of the γ-calculus [Smo94] with constraints and search. The γ-calculus shares many similarities with the π-calculus: a composition of γ terms, written $E_1 \wedge E_2$, and a declaration $\exists uE$, are similar to the parallel composition and restriction of the π-calculus. The primitives of γ-calculus include *abstraction* $a : \bar{x}/E$ (with $\bar{x}$ is linear, ie. consist of pairwise distinct variables) and *application* $a\bar{u}$, with the following reduction.

$$a\bar{u} \ \wedge \ a : \bar{x}/E \quad \rightarrow \quad E[\bar{u}/\bar{v}] \ \wedge \ a : \bar{x}/E \quad \text{if } \bar{u} \text{ is free in } E \text{ and } |\bar{u}| = |\bar{v}|$$

The reduction is obviously akin to a reduction rule for a replicated input (cf. (Pi-Replic) in Figure 2.2). Victor and Parrow [VP96] gave an encoding of the γ-calculus into the π-calculus, and established an operational correspondence between the reductions in the γ-calculus and its encoding.

8.1.3 Discussion

First of all, we summarise the design choices of the reviewed calculi in Table 8.1. We use abbreviations *LT, LA* for location-transparent, location-aware interaction, *NU* for non-unique (naming), and *barbed* for barbed bisimulation equivalences [MS92], *bisim* for standard bisimulation and *asyn bisim* for asynchronous bisimulation [ACS98].

The design choices described provide different levels of abstraction, each of which is suitable for modelling particular types of protocols or applications. However, we argue that in order to use a calculus for expressing distributed applications over wide-area networks, and for such applications to be implementable, the choices involved must be supported by the standard network technology and not rely on a complex distributed infrastructure. Many of the design choices discussed require infrastructure support. In location-transparent interaction, outputs on a channel c on an agent must be able to interact with an input on c on any agent in a fair manner. This requires a handshaking procedure whenever an agent wishes to input or

	distribution		mobility	interaction	op. equivalence
	naming	loc. struct.			
π_l	unique	flat	spawning	LT	barbed
π_{1l}	unique	flat	spawning	LT	barbed & bisim
$\mathrm{D}\pi_1^{\mathrm{r}}$	unique	flat	spawning	LA (LI)	asyn bisim
distr. CCS	unique	flat	spawning	LT	LF
Dπ	unique	tree	migration	LA (LI)	barbed
D$\pi\lambda$	unique	flat	spawning	LT	none
dpi	unique	tree	migration	LT	none
D-Join	unique	tree	migration	LT	barbed
Seal	NU	tree	migration	LA (LD)	barbed
Ambient	NU	tree	migration	LA (LD)	barbed
Box-π	NU	tree	none	LA (LD)	asyn bisim
distr. TyCO	unique	flat	spawning	LA (LI)	none
Nomadic π	unique	two-level	migration	LA (LD & LI)	bisim

Table 8.1: Design choices of distributed process calculi

output on a channel. Since agents are capable of moving between sites, the infrastructure for supporting this can be highly complex — having to track movement of agents as well as ready-to-interact inputs and outputs.

The join-style interaction, though location-transparent, is less costly since there exists a unique location which is capable for receiving from each channel. Nevertheless, each output needs to be moved to the location where it may react with inputs. An infrastructure is required to support this, since a map of where each channel is defined must be maintained. Moreover, since the Join language supports structured migration, the infrastructure must ensure that, while an output is forwarded to its destination, neither the target location nor any of its ancestors migration away. These requirements are likely to make infrastructure complex and difficult to verify.

Location-dependent communication primitives, on the other hand, are supported by the standard network technology. The Seal, Ambient, low-level Nomadic π and Box π-calculi, which only employ LD primitives, can therefore be used as convincing bases for programming languages for wide-area networks.

Nevertheless high-level primitives (those which require support from distributed infrastructure) are convenient for writing applications. Moreover, specifications can be given in a language which supports *both* high- and low-level primitives. This allows implementations (expressed purely using low-level primitives) to be verified by comparing their behaviour with the specifications.

From a theoretical point of view, comparing the design choices of Nomadic π to other choices reveals the following.

- Location-dependency makes congruences — operational equivalences which are preserved under parallel composition and **new**-binding — difficult to obtain. In distributed process calculi which provide only location-independent primitives, an operational congruence generally coincides with a standard notion of bisimulation, as has been shown for the distributed join calculus (in absence of failure) in [BFL98], and for π_{1l} calculus in [Ama97]. This is not the case for calculi offering location-dependent primitives. Gordon and Cardelli gave a definition of an operational congruence for the Ambient calculus [GC99]. Such a definition, however, involves quantification over program contexts, which can make proving congruence relationship between processes difficult. Congruences offered by Nomadic π — strong, weak and expansion congruences (see Chapter 5) — do not involve quantification over program contexts, but still require a considerable extension to the standard notion of bisimulation.

- The complex migration mechanism offered by structured migration could make the definition of temporary immobility (see Section 6.4) complex, and subsequently make proofs of temporary immobility properties difficult.
- It would be possible to adopt the join-style of channel definition, ie. ensuring that there is a unique agent capable of reading from each channel, as in $\mathrm{D}\pi_1^{\mathrm{r}}$. This would simplify many results in Section 6.2, since certain channel usage disciplines are respected by processes using join patterns. However, restricting channels in this way can make infrastructure translations more complex. For example, in the $\mathcal{C}$-translation, the `Deliverer` process must be specific for each agent, since the name `deliver` can now be used for input at a single agent. This means that the daemon must explicitly keep a map of what channel it must use for forwarding messages to each agent.

For these reasons, we did not adapt the high-level primitives of structured migration and join-patterns.

8.2 I/O Automata

The input/output (I/O) automaton model [LT87, LT88] is one of labelled state transition systems, intended for modelling reactive systems interacting with the environment. Originally developed for specifying and verifying theoretical distributed algorithms, I/O automata have also been applied to practical communication services such as TCP, distributed shared memory, and group communication (see [GLV97] for many references). It has helped in resolving ambiguities and contributing proofs that such systems meet their specifications. In this section, we shall briefly describe the model, some of its proof techniques, and make some comparisons between the model and Nomadic π-calculi. Here we refrain from giving examples, many of which can be found in [Lyn96].

An I/O automaton is a simple type of state machine in which the transitions are associated with named *actions*. Such actions can be classified as either *input*, *output* or *internal*. The input and output actions are used for communicating with the automaton's environment, whereas internal actions, like τ actions of the π-calculus, are discarded in defining compositions. More formally, each automaton A consists of five components: a set of states $states(A)$, a non-empty set of start states $start(A) \subseteq states(A)$, a set of actions $acts(A)$, a transition relation, and a set of tasks, each of which is a non-input action. The transition relation of an I/O automaton is similar to that of the π-calculus in that it defines transitions between states and actions which are committed as a result of such transitions. For example, if states s, s' can be related by a transition relation, written $s \xrightarrow{\pi} s'$, then s can evolve to s' by committing the action π. One of the fundamental assumptions made in the model, however, is that an automaton is unable to block its input. This comes from the intuition that input actions are not under the control of the automata, but rather from the outside world. The transition relations must therefore define transitions made by every input action from every state — a condition which is not required by the transition relation of the π-calculus. An *execution* of an automaton A is a (possibly infinite) sequence of alternating states and actions $s_0\pi_0, s_1\pi_1, \ldots$ such that s_0 is a start state in $start(A)$ and, for all $i \geq 0$, $s_i \xrightarrow{\pi_i} s_{i+1}$ belongs to the transition relation of A. The set of tasks can be thought of as an abstract description of "task" or "threads of control" within an automaton. It is used for defining fairness conditions on execution of the automaton, ensuring that the tasks are given fair turns during execution.

A complex automaton can be obtained by combining simpler automata using a *parallel composition* operator. An action of a composite automaton is generated from one of its components, and if several components may commit the same action, they may synchronise (ie. simultaneously make transition under such an action). A pair of components A and A' may "interact" if they are able to commit an action π which is an input action for A and an output action for A' (or vice versa) — by performing π, the two components simultaneously undergo a transition. In order that the composition operator will enjoy nice theoretical properties, component automata must share no output actions, and their internal actions must not be actions of any other automata. The *hiding* operation may be applied to an automaton to hide some of its output actions by reclassifying them as internal actions.

Verification and Proof Techniques In addition to modelling reactive systems, I/O automata can be used for reasoning about their behaviour. A specification can be given in the form of *trace properties*, ranged over by P, consisting of pairs of a set of external actions (ie. input or output actions) and a set of sequences of such external actions. An automaton A is said to satisfy a trace property P if its external actions and its traces (ie. observable prefixes of executions of A) are included within those of P. Conditions can be imposed on trace properties so that they satisfy *safety* or *liveness* conditions [Lam77], informally ensuring that some "bad" thing never happens or that some "good" thing will eventually happen. To prove that an automaton satisfies a trace safety property or a trace liveness property, the following techniques can be employed.

- **Simulation** It can be proved that if an automaton A *simulates* another automaton B then all the traces of B are traces of A. There exists several notions of simulation [LV94]. Many of these share similarities with operational relations in the literature of process calculi.

- **Modular decomposition** The behaviour of a composite automaton can be reasoned about via the behaviour of its components. It can be shown that if each component A_i satisfies trace property P_i then the composite automaton $\prod_i A_i$ satisfies the composite trace property $\prod_i P_i$ obtained by interleaving traces of P_i. Additional preconditions are required to ensure that the composite trace satisfies eg. safety conditions.
- **Hierarchical decomposition** A system or an algorithm can be described by a hierarchy of automata, each at different levels of abstraction. For example, the highest level may be a specification, whereas the lower-level automata may look more like actual systems or algorithms that will be used in practice. The best way of proving properties of the lower-level automata is often by relating them to automata at higher levels in the hierarchy, rather than carrying out the proof from scratch.

These techniques have been successfully applied in proofs to practical algorithms such as a shared memory service [FKL95], group communication [FLS98, Cho97] and a standard communication service [Smi97]. Although most of the verification of quoted protocols have been done by hand, it is claimed that much of such proofs (especially simulation proofs) is sufficiently stylised to admit computer assistance using interactive theorem provers such as the Larch Prover (LP) [GG91].

IOA Language One of the main problems of using I/O automata is, quoting Garland, Lynch and Vaziri [GLV97], that "there exists no formal connection between verified designs and the corresponding final code." Following this, the IOA language [GLV97], a formal language based on I/O automata, has been designed. The language is to be equipped with a coordinated suite of tools to support the development and analysis of IOA programs. These tools (will) include analysis tools, ranging from simulation to model-checking to theorem-proving tools. They (will) also include tools for translating IOA programs into source code in an existing programming language (eg. Java), thereby producing an executable code for a distributed system. Much of the design and implementation of these tools is under current research.

Discussion The style and syntax of I/O automata is very different to that of process calculi. States of an automaton can be any arbitrary mathematical objects, whereas states of process calculi are implicit in process terms, which are algebraic expressions rigorously constructed from a small set of operators. The style of interaction of I/O automata model (via input, output and internal actions) is similar to that of process calculi. However, an I/O automaton definition of a labelled transition relation is much more arbitrary than that of process calculi. The arbitrariness of definitions of I/O automata provides flexibility, allowing a wide range of programs, protocols and algorithms to be expressed; however, such definitions often require further descriptions, commonly via natural language, which are not part of the model.

The proof techniques of I/O automata model are somewhat akin to those of the process calculi, due to similarity in style of operational semantics. In process calculi, trace properties can be expressed as process terms; to verify that a process satisfies a trace property, we may prove that such process terms are related by some operational equivalence or preorder. Some of these notions of equivalences ignore internal actions, which ensures that only observable actions are compared between processes. The technique of modular decomposition can also be applied to process calculi, provided that the operational equivalence used is preserved by parallel composition. In fact, if such an equivalence is a congruence than decomposition is not limited to parallel composition of restricted components (as in the I/O automata model) but any parallel composition as well as other operators and prefixes. The technique of hierarchical decomposition is also used in the context of process calculi, for example, to prove that two calculi are equally expressive, an intermediate calculus might be introduced so as to reduce the complexity of the proof (see eg. [NP96] and Chapter 7 of this thesis).

The I/O automata model lacks an inherent notion of types. This can be somewhat disadvantageous, since some types have been shown to provide powerful proof techniques (see eg. [KPT96, San99, IK01]). The IOA language introduces some simple datatypes such as boolean, integer, real number and string, as well as some type constructors such as array, set, multiset and map together with associated operators. This type system, however, seems to aim at preventing runtime errors rather than providing useful proof techniques.

Like the π-calculus, the I/O automata model has no clear notions of distribution nor locality. Networks are modelled as a directed graph with an automaton `CO(i)` corresponding to each node `i` and a channel `channel(i, j)` (also an automaton) corresponding to each edge from node `i` to node `j`. Unlike the π-calculus, an I/O automaton cannot dynamically generate new names or exchange names over channels. This makes its network model static and unable to directly cope with dynamic reconfigurations without some explicit encoding. Furthermore, since an automaton can neither move from one node to another nor transmit an automaton to be executed at the other node, it is not possible to directly model mobile computation using the I/O automata model.

8.3 Mobile UNITY

Mobile UNITY [RMP97] is an extension of UNITY [CM88] to address the problem of modelling dynamically reconfiguring distributed systems and disconnections of their components. Additionally, it attempts to address design issues raised by mobile computing (in particular, device mobility — see Section 1.1), due to the characteristic of the wireless connection and the nature of application and services that will be demanded by users of the new technologies. Three such issues are *decoupling*, the ability of application to run while disconnected from or weakly connected to servers; *context dependencies*, the fact that application behaviour might depend on the totality of the concurrent context, including the current location and the nearness of other components; and *location transparency* provided by some distributed infrastructure such as Mobile IP. Mobile UNITY has been used in an exercise involving the specification and verification of Mobile IP [MR99], and in modelling various forms of program mobility [PRM97].

Chandy and Misra's UNITY model [CM88] is based on automata that interact via shared variables instead of shared actions. In UNITY, a system may consists of several programs which share identically named variables. Each program consists of three sections: a **declare**, an **initially** and an **assign** section. The **declare** and **initially** define variable types and the initial program conditions (eg. the initial values of program variables). The **assign** section is simply sets of assignment statements which execute atomically and are selected for execution in a weakly fair manner, ie. in an infinite computation each statement is scheduled for execution infinitely often. Mobile UNITY model extends statement constructs with *transaction*, *inhibitor* and *reactive* statements. A transaction provides a form of sequential execution without intervention of any other statements from the same program. It is of the form $l :: \langle s_1; \ldots; s_n \rangle$, where each s_i is an assignment statement, and l is the (optional) label of such a statement. An inhibitor **inhibit** n **when** p prevents the statement labelled n from execution whenever the predicate p holds. Finally, a reactive statement s **reacts–to** p behaves as a while statement (ie. if the predicate p holds, the statement s is repeatedly executed until p is falsified). Additionally, a reactive statement has a higher priority than other non-reactive statements in the program (ie. all the reactive statements must finish their execution before any other statement may proceed).

To model concurrent systems that contain dynamically reconfiguring components, Mobile UNITY considers each program a unit of mobility, which is accommodated by attaching a distinguished location variable to each program. This location variable, conventionally λ, provides both location awareness and location control to the individual program. There is no restriction on the type of location variable λ, it can therefore be discrete or continuous, single- or multi-dimensional, or it might correspond to latitude and longitude for a physically mobile platform, or it may be a network or memory address for a mobile agent. Movement of a program between locations (under its own control or not), is modelled by assignment of new values to the variable modelling its location. This can be interpreted as actual physical movement in the systems where agents are capable of moving, or, in a system of mobile agents, the execution of this assignment would have the effect of migrating an agent to a new site. Note that λ may also appear on the right-hand side of some assignment statements if there is any location-dependent behaviour internal to the program.

A Mobile UNITY system can be declared by giving the programs of its components (together with their initial locations), and specifying how such programs interact in an **Interactions** section. Programs may interact when they are in close proximity. In the most primitive form, an "interaction" may happen if one program accesses variables of another program which is at the same location. Sophisticated forms of interactions are achieved by two novel constructs: *transient variable sharing* and *transient action synchronisation*, which can be expressed as syntactic sugar of more primitive Mobile UNITY constructs. The former construct allows a variable owned by one program to be shared in a transparent manner with different programs at different times depending upon their relative locations in space. The latter construct allows a statement owned by one program to be executed in parallel with statements owned by other programs when certain spatial conditions are met.

Verification and Proof Methods Rather than dealing with execution sequences (as in the process calculi and the I/O automata model), the formal semantics of UNITY is given in terms of program properties that can be deduced from the text. Program properties may be stated and may be verified using the Mobile UNITY proof logic. Basic safety (expressed as **unless** or **co**) and liveness (expressed as **ensures**) properties are proven by quantifying over the *state predicates*, each of which can be constructed from variable names, constants, mathematical operators, and standard boolean connectives. Informally, p **co** q means that if the program is in a state satisfying p then the next state after any assignment is executed must satisfy q; **transient** p means that there exists any assignment whose execution falsifies the predicate p. The formal definitions of these two constructs can be expressed in first order logic and together with Hoare-triple notion $\{p\}s\{q\}$ [Hoa69]; here we omit these definitions. Using these notations and proof logic, the correctness of programs using mobile code can be verified (see some examples in [PRM97]).

The proof rules of Mobile UNITY are developed from those of UNITY. The new constructs (reactive, inhibitor and transaction statements) requires redefinition of the basic notion of transition. Proofs of properties of UNITY programs can be mechanised using tools such as HOL-UNITY [FKJ93], Coq-UNITY [HC96] and Isabelle [Pau99]. The authors of Mobile UNITY have not mentioned the possibility of using these tools to mechanise proofs of Mobile UNITY program properties.

Lime Lime (Linda in a Mobile Environment) [PMR99] is a system designed to assist in the rapid development of dependable mobile application over both wired and ad-hoc networks. It is an extension of Linda [Gel85] to a mobile environment. In Linda, coordination is achieved through a *tuple space*. Components can access the tuple space by inserting, reading, or withdrawing tuples containing information — independently of their actual location, The model intrinsically provides both spatial and temporal decoupling, making it a suitable candidate for a programming language whose formal semantics can be based on Mobile UNITY.

In the model underlying Lime, mobile agents are programs that can move between mobile hosts, which may also move in the network. Tuple spaces are permanently bound to mobile agents and mobile hosts. These can be (transiently) shared when agents or hosts are co-located. Operations on tuples are location-aware, allowing programmers to specify the tuple space of an agent in which a tuple t shall be inserted into or removed from.

Since Lime deals both with movements of programs as well as of physical machines, the technical issues involved in its development are highly complex. A prototype of Lime, building upon IBM's T-Space [WMLF98], remains under development.

Discussion It is difficult to compare the Mobile UNITY model with process calculi, for their style and verification techniques differ greatly. Interaction between Mobile UNITY programs, for instance, is based on transient sharing of variables when "agents" are in proximity, whereas in the π-calculus, interaction is based on message passing. The biggest differences between the two models is perhaps the style of semantics: the axiomatic style of Mobile UNITY model and the operational style of the process calculi. Owing to their axiomatic semantics, properties of Mobile UNITY programs can be more concise than those for process calculi; and therefore can be easier to grasp as well as to prove. In general, however, axiomatic semantics, although easier to define mathematically, can abstract away too many details, making it harder to confirm that such a definition corresponds to one's intuition (eg. it is not clear how Mobile UNITY systems interact with users via I/O devices). This is less problematic for an operational semantics, since any of its rule obviously expresses a transformation of a particular program state (or process). A drawback of working with program states is that it is harder to grasp high-level descriptions, eg. program invariants.

Comparing the expressiveness of the two models is also difficult, although we may note that in Mobile UNITY the set of components making up a system is fixed, whereas in most variants of distributed π-calculus, agents can be dynamically created. On the other hand, the Mobile UNITY model takes into account disconnected operations and other aspects relating to mobile-device networks, which remains largely unaddressed in the process calculi community.

Chapter 9

Conclusions and Future Work

In this chapter we summarise the work done within this thesis and its contribution to computer science. We conclude with some discussion of future work.

9.1 Summary

Wojciechowski and Sewell [WS99, Woj00a] have shown that Nomadic π-calculi can form a basis for a distributed programming language design. Furthermore its two-level architecture allows the distributed infrastructure algorithms to be expressed concisely and precisely as translations from high-level to low-level Nomadic π-calculi. In this thesis, we turned to the problem of how such infrastructures can be verified. We have rigorously developed the Nomadic π-calculi (starting from [SWP99] which gave only a preliminary syntax and reduction semantics), giving it a type system, a labelled transition semantics, operational relations and proof techniques. This development allowed us to state and prove the correctness of infrastructures — as a substantial example, we proved that a central-forwarding-server algorithm is correct w.r.t. coupled simulation.

Many of the semantic theories and proof techniques developed are drawn from those of process calculi, adapted to the distributed setting with mobile agents. The techniques we adapted are: input/output subtyping [PS96], coupled simulation [PS92], uniform receptiveness [San99], techniques of bisimulation "up to" [SM92], and the strategies employed in Nestmann's correctness proofs of choice encodings [Nes96, NP96]. Adapting these techniques was non-trivial, for the new notions and primitives often introduced unexpected difficulties. For example, as discussed in Sections 6.2.2-6.2.3, although in the π-calculus access restrictions are sufficient for deriving an expansion from a functional step (Lemma 6.2.3), to obtain such a result in Nomadic π, we also needed uniform receptiveness. This is for dealing with dynamically created agents, since they may introduce new function definitions.

We have also developed novel techniques for dealing with mobile agents, most notably translocating equivalences and the technique of temporary immobility. The former extends the standard definitions of operational equivalences so that they take into account spontaneous movements of agents by the environment. This allows compositional reasoning to be applied to proofs involving operational equivalences. The latter captures the intuition that while an agent is waiting for an acknowledgement or a lock somewhere in the system, it may not migrate. We made this precise and showed that a deterministic reduction, when placed in parallel with a temporarily immobile process, gives rise to an expansion. Furthermore, we developed a technique (akin to the technique of "up to" equivalence) for proving processes temporarily immobile. We illustrated such a technique by showing the temporary immobility of agents in various states of the $\mathcal{C}$-encoding.

The actual proof of correctness of the example infrastructure was non-trivial. Despite the simplicity of the algorithm employed, many τ-steps are introduced by the encoding, inducing a large number of intermediate states. This makes it impossible to transcribe a direct operational equivalence relation between the source program and its encoding. Here we adopted Nestmann's strategy (cited above) of introducing an intermediate language which helps manage the additional τ-steps. The definition of such a language is more complex than that of Nestmann, as our encoding is non-uniform; it therefore requires the states of all agents (as well as that of the daemon) to be included in the syntax.

The current correctness statement does not take divergence into account, but we believe this could be easily addressed.

9.2 Future Work

There are many directions in which the work done in this thesis can be developed and applied further.

Proving other infrastructures Despite its simplicity, the example infrastructure employs several programming techniques commonly used in designing distributed algorithms, such as locking and synchronisation. In this thesis, we have devised some semantic and proof techniques which are demonstrably capable of handling those programming techniques, at least in a simple algorithm. We believe that more sophisticated algorithms can be dealt with using the same techniques, albeit with new intermediate languages (tailored to particular algorithms).

To support this point, we have sketched an intermediate language for proving the correctness of the forwarding-pointers infrastructure given in [SWP99]. The original encoding is first modified, in the same way as our modified version of the central-forwarding server translation in the cited work — adding types, using fresh channels for acknowledgements, and extending the top-level translation to arbitrary located processes. The modified encoding and the sketch of the intermediate language is given in Appendix A.

Types Although the type system given in Chapter 3 is sufficiently rich (with polymorphism and subtyping) for expressing various distributed infrastructures, it can be improved further. The Nomadic Pict language has a rich type system, inherited from Pict, which is useful for programming distributed and mobile applications. To avoid meta-theoretic complexity however, many typing features supported in Pict are not included in the type system of this thesis — among features omitted are recursive types, dynamic types, variant types, and extensible records. Recursive definitions requires recursive types — the omission of such types therefore decreases the expressiveness of Nomadic π-calculi, for many algorithms require recursive definitions. We deliberately avoided uses of recursion in the definition of finite maps, given in Section 6.5, which results in an inefficient implementation in terms of storage (for out-of-date entries are never garbage-collected). In general, however, uses of recursion or recursive types are inevitable. Dynamic and variant types can also be useful for programming, since they allow an agent to interact with another agent, even though the type of the data exchanged cannot be determined at compile time.

Improvement of the existing type system could also provide some powerful proof techniques, eg. by introducing typing features such as linear types [KPT96], uniform receptiveness [San99], and sendability [YH99b]. Again to avoid meta-theoretic complexity, we replaced these types by syntactic analysis (see Section 6.2). Inclusion of such features in the type system would make the relevant lemmas simpler to state and prove. Furthermore, properties such as "`deliver` is uniformly receptive in $\mathcal{C}[\![LP]\!]$" could be obtained by typechecking alone.

Recently Igarashi and Kobayashi proposed a generic type system for the π-calculus [IK01], where types and type environments are expressed as abstract processes. They also showed that various non-trivial type systems (such as those for ensuring the absence of race conditions and deadlock) can be obtained as instances of the generic type system by changing the subtyping relation and the consistency condition. They also showed that several important properties (eg. the subject reduction) of these particular type systems can be obtained independently of the instantiated conditions. Adapting the generic type system to Nomadic π-calculi could offer many advantages. Firstly, such a system integrates many existing type systems, including the typing features cited above, and is easy to extend. Secondly, expressing types and type environments as abstract processes allows various properties of a process to be checked by verifying the corresponding properties of its type environment. It would be interesting to investigate whether, for example, the temporary immobility property of processes can be verified in this way.

Proof techniques The semantic and proof techniques in this thesis can be developed further so that properties are easier to state, prove, and reuse. Previously we suggest extending the type system with linear types, uniform receptiveness and sendability. Another improvement is extending the definition of operational equivalences and preorders to basic processes. We naturally expect translocating equivalences for basic processes to be congruences, allowing decomposition by other prefixes in addition to parallel composition and **new**-binders.

It is conceivable that modal logic [HM85] (see also brief description in Section 8.1.1 on page 162) may play a major role in distributed infrastructure verification. Indeed the definition of temporary immobility could be given in terms of a modal logic, and proofs of processes being temporarily immobile could be derived using logical rules. Modal logic may also be required for stating robustness properties, for example, "provided that such and such machines do not fail simultaneously, this infrastructure correctly provides location-independent communication". There are several known difficulties though; most important of all is the problem of compositionality: how can the temporal properties of a process $P|Q$ be inferred, automatically or manually from those of its components P and Q. Many solutions have been proposed, although one of the most promising is the approach of Dam applied to the CCS [Dam95], the π-calculus without restriction [AD94], and the finite state π-processes [Dam01]. Further research is required to fully apply this approach to Nomadic π-calculi.

Automated verification All the lemmas in this thesis have been verified by hand. This is feasible since the verified algorithm is not too complex. Realistic algorithms, however, are likely to be much more complex, with more states — to prove these algorithms correct, we need some automated tools. Many such tools for process calculi exist (see an overview in Section 8.1.1 on page 162), but most can only be used for verifying *finite state* processes (ie. those which do not admit parallel composition within recursively defined processes or replicated processes). Distributed infrastructures and applications, however, are unlikely to be finite state (since services in wide-area networks are generally persistent); this means that these tools cannot be used for verifying infrastructures.

It is also perhaps impossible for a machine to decide what kind of intermediate language is suitable for a particular algorithm, and to analyse (potentially infinite) transition sequences. Our experience, however, shows that many part of the correctness proof — especially the proof that an intermediate language term is related by an expansion to its image in the source language — involves simplifying a complex process by series of conditional *rewriting* (ie. by substituting its subprocesses by simpler subprocesses, which are related to them by expansions whenever some conditions are met). An alternative is therefore to treat verified results of our proof techniques as *rewriting axioms*. For example, a process $@_b\langle b@s\rangle c\,!\,v$ can be substituted with $@_b c\,!\,v$ whenever it is placed in the context where the agent b is located at s, and b is temporarily immobile such a context — the result of this rewriting is expanded by the original process (by Lemma 6.4.2). These "rewriting axioms" can be verified by hand since their proofs are generally not too complex. By embedding such rewriting axioms in a theorem-prover, the proofs of complex results (such as those similar to Lemma 7.3.5) could be partly done automatically. Furthermore, the theorem-prover could help managing the complex premises involved in such rewriting.

Observational characteristics The correctness result poses a problem: how can we be sure that the operational relation chosen for the correctness statement is appropriate? In [Sew97], Sewell considered an example application of the π-calculus, and investigated how far it is possible to argue, from facts about the application, that some model of process algebra is the most appropriate. This involved defining a notion of observation that can be seen to be appropriate for the programming language Pict. Such a notion takes into account the interactions between an actual Pict implementation and a user, together with their relationship to the structured operational semantics.

Similar investigation could be conducted for applications of the Nomadic π-calculi. The interactions between a Nomadic Pict implementation and users could be more complex, since users as well as programs can be distributed over the network. We have briefly discussed how users and agents interact in Section 5.4.2; such a discussion, however, did not address many considerations such as termination, divergence, and fairness.

Security In this thesis, the high-level calculus supports location-independent communication between agents. We may also consider other high-level primitives such as group communication, multicast, join-style communication, structured migration and secured inter-agent communication. Security in particular has been a major concern for the use of mobile agents. A number of research has emerged during the recent years on adding some notions of security to process calculi [AG97, CG98, RH99, VC99, SV99, AF01]. Nomadic π contains no such notions. To the low-level calculus, we may add cryptographic primitives (as in the Spi calculus [AG97]), allowing definitions of infrastructures which provide secured communication (see also [AFG98]); or we may add some notion of *trust* [RH99] to the type system, allowing eg. uses of private resources to be prevented from malicious 'untrusted' agents.

Failure semantics To express and reason about distributed infrastructures which are robust under some form of failure, or to support disconnected operations, new primitives for detecting possible failure are required. Wojciechowski et al. [WS99, Woj00a] added a single *timed input* primitive with timeout value n, written

$$\texttt{wait}\ c\texttt{?}p \rightarrow P,\ n \rightarrow Q$$

to the low-level calculus. The informal semantics of this is that: if a message on channel c is received within n seconds then P will be started as in a normal input, otherwise Q will be. The timing is approximate as the runtime system may introduced some delays. The operational semantics of this requires the existing configuration to be extended with a global time UTC (Coordinated Universal Time) t, and a primitive $\texttt{wait}_t\ c\texttt{?}p \rightarrow P, Q$. We quote the reduction rules (from [Woj00a]) below.

$$\begin{array}{rcl}
\Gamma, t, @_a\texttt{wait}\ c\texttt{?}p \rightarrow P,\ n \rightarrow Q & \rightarrow & \Gamma, t+1, @_a\texttt{wait}_{t+n}\ c\texttt{?}p \rightarrow P, Q \\
\Gamma, t, @_a\texttt{wait}_{t_n}\ c\texttt{?}p \rightarrow P, Q & \rightarrow & \Gamma, t, @_a Q \quad \text{if } t \geq t_n \\
\Gamma, t, @_a c\texttt{!}v \mid \texttt{wait}_{t_n}\ c\texttt{?}p \rightarrow P, Q & \rightarrow & \Gamma, t+1, @_a \mathsf{match}(p, v) P
\end{array}$$

The use of timeout primitive requires no particular infrastructure and can therefore be (and has been) implemented. Some other models, such as the π_{1l} [AP94] and the distributed join calculus [FGL$^+$96], support reliable failure detectors, allowing programmers to detect location or site failures and take some action. It has been shown [FLP85], however, that distributed consensus (such as agreeing on which sites have failed) cannot be achieved in a system consisting of a collection of asynchronous processes. In practice, a good approximation can be achieved [CT96], but the algorithms required can be costly and hard to reason about.

Reasoning about failures and disconnected operations using this timeout primitive poses many challenges: for example, how do we state robustness properties, how do we handle time in the labelled transition semantics and operational equivalences, and how do we define operational equivalences which are congruences in this setting. The QSCD algorithm

[Woj00a], which supports disconnected operations, may serve as a good starting point for further research in this area.

9.3 Conclusion

We have demonstrated that expressing infrastructure algorithms in Nomadic π-calculi offers three advantages. Firstly, the representations of the algorithms are concise, for concurrency, asynchronous message passing and name generation are primitives of the calculi. Secondly, the informal description of the calculi and of the algorithms are precisely captured by the operational semantics of the calculi. Moreover, the operational semantics can be used for formal reasoning. Finally, such algorithms, as well as applications written in Nomadic π, can be rapidly prototyped using the Nomadic Pict language.

For many, the Internet (and wide-area networks in general) is becoming an indispensable part of everyday life. Many technologies for wide-area distributed systems are emerging for coping with the ever-increasing demand for novel network applications and services. Apart from the mobile agent technology addressed in this thesis, other technologies, such as ubiquitous computing [Wei93], ad-hoc wireless networks [Gro], and active networks [TSS$^+$97], have been recognised as promising. They have been widely studied and deployed. As demonstrated in this thesis, formal semantics does not only enhance our understanding of complex network technology, but can also give us some confidence in its correctness and reliability. We hope this thesis contributes to the view that rich network technologies should be built upon strong semantic foundations.

Appendix A

A Forwarding-Pointers Infrastructure Translation

In this appendix, we give a forwarding-pointers infrastructure based on the definition given in [SWP99], slightly modified with exact types and fresh acknowledgements. These added type annotations have been checked with the Nomadic Pict type checker (although this does not check the static/mobile subtyping). This algorithm is more distributed than the central-forwarding-server algorithm, since it employs daemons on each site for maintaining chains of forwarding pointers for agents which have migrated. We describe a sketch of an intermediate language which could be used for proving this infrastructure correct. The reader may refer to [SWP99] for detailed explanation of the algorithm.

The daemons are implemented as static agents; the translation $\mathcal{FP}_\Phi\,[\![LP]\!]$ of a located process $LP = \mathbf{new}\ \Delta\ \mathbf{in}\ @_{a_1}P_1 \mid \ldots \mid @_{a_n}P_n$, well-typed w.r.t. Φ, then consists roughly of the daemon agent at each site in parallel with a compositional translation $[\![P_i]\!]_{a_i}$ of each source agent. Assuming distinct $s_1, \ldots, s_m$ are all the sites in Φ, we define $\mathcal{FP}_\Phi\,[\![LP]\!]$ as follows:

$$
\begin{array}{l}
\mathbf{new}\ \texttt{register},\texttt{migrating} : \texttt{\^{}}^{\texttt{rw}}[\texttt{Agent}^{\texttt{s}}\ \texttt{\^{}}^{\texttt{w}}[]],\\
\quad \texttt{migrated} : \texttt{\^{}}^{\texttt{rw}}[\texttt{Agent}^{\texttt{s}}\ [\texttt{Site Agent}^{\texttt{s}}]\ \texttt{\^{}}^{\texttt{w}}[]],\\
\quad \texttt{message} : \texttt{\^{}}^{\texttt{rw}}\ \{X\}\ [[\texttt{Agent}^{\texttt{s}}\ \texttt{Site Agent}^{\texttt{s}}]\ \texttt{\^{}}^{\texttt{w}}X\ X],\\
\quad \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}[\texttt{Site Agent}^{\texttt{s}}], \texttt{lock} : \texttt{\^{}}^{\texttt{rw}}\texttt{Map}[\texttt{Agent}^{\texttt{s}}\ \texttt{\^{}}^{\texttt{rw}}[\texttt{Site Agent}^{\texttt{s}}]]\\
\quad D_1 : \texttt{Agent}^{\texttt{s}}@s_1, \ldots, D_m : \texttt{Agent}^{\texttt{s}}@s_m\\
\mathbf{in}\\
\quad @_{D_1}(\mathit{Daemon}_{s_1} \mid \texttt{lock!}\mathit{map}) \mid \ldots \mid @_{D_m}(\mathit{Daemon}_{s_m} \mid \texttt{lock!}\mathit{map})\\
\quad \mid @_{a_1}[\![P_1]\!]_{a_1} \mid \ldots \mid @_{a_n}[\![P_n]\!]_{a_n}
\end{array}
$$

$Daemon_s$

$\stackrel{\text{def}}{=}$ **let** $[S\ DS] = s$ **in**

*register?$[B$ rack$]\rightarrow$lock?$m\rightarrow$

  lookup[Agent$^{\mathtt{s}}$ [Site Agent$^{\mathtt{s}}$]] B **in** m **with**

    found(Bstate)$\rightarrow$

      Bstate?[_ _]$\rightarrow$

        Bstate!$[S\ DS]$ | lock!m | $\langle B\rangle$rack![]

    notfound$\rightarrow$

      new Bstate : ^$^{\mathtt{rw}}$[Site Agent$^{\mathtt{s}}$] **in**

        Bstate!$[S\ DS]$ | $\langle B\rangle$rack![]

        | **let**[Agent$^{\mathtt{s}}$ [Site Agent$^{\mathtt{s}}$]] $m' = (m$ **with** $B \mapsto$ BState) **in**

          lock!m'

| *migrating?$[B$ mack$]\rightarrow$lock?$m\rightarrow$

  lookup[Agent$^{\mathtt{s}}$ [Site Agent$^{\mathtt{s}}$]] B **in** m **with**

    found(Bstate)$\rightarrow$

      Bstate?[_ _]$\rightarrow$(lock!m | $\langle B\rangle$mack![])

    notfound$\rightarrow$**0**

| *migrated?$[B\ [U\ DU]$ ack$]\rightarrow$lock?$m\rightarrow$

  lookup[Agent$^{\mathtt{s}}$ [Site Agent$^{\mathtt{s}}$]] B **in** m **with**

    found(Bstate)$\rightarrow$

      Bstate?[_ _]$\rightarrow$(lock!m | $\langle B@U\rangle$ack![] | Bstate!$[U\ DU]$)

    notfound$\rightarrow$**0**

| *message? $\{X\}$ $[[B\ U\ DU]\ c\ v]\rightarrow$lock?$m\rightarrow$

  lookup[Agent$^{\mathtt{s}}$ [Site Agent$^{\mathtt{s}}$]] B **in** m **with**

    found(Bstate)$\rightarrow$

      Bstate?$[R\ DR]\rightarrow$lock!m

        | **iflocal** $\langle B\rangle c!v$ **then** Bstate!$[R\ DR]$

          else $\langle DR@R\rangle$message! $\{X\}$ $[[B\ U\ DU]\ c\ v]$

            | Bstate!$[R\ DR]$

    notfound$\rightarrow$lock!m

      | $\langle DU@U\rangle$message! $\{X\}$ $[[B\ U\ DU]\ c\ v]$

Figure A.1: Forwarding-pointer: the local daemon

$$
\begin{array}{l}
[\![\langle b@?\rangle c\,\mathtt{!}\,v]\!]_a \\
\quad \overset{\text{def}}{=}\ \ \mathtt{currentloc?}[S\ DS] \rightarrow \\
\qquad\quad \langle D@SD\rangle\mathtt{message!}\,\{T\}\,[b\ c\ v] \\
\qquad\quad \mid \mathtt{currentloc!}\,[S\ DS] \\
\\
[\![\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q]\!]_a \\
\quad \overset{\text{def}}{=}\ \ \mathtt{currentloc?}[S\ DS] \rightarrow \\
\qquad\quad \mathbf{new}\ \mathtt{pack} : \hat{}^{\mathtt{rw}}[], \mathtt{rack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \\
\qquad\quad \mathbf{create}^Z\ B = \\
\qquad\qquad \mathbf{let}\ b = [B\ S\ DS]\ \mathbf{in} \\
\qquad\qquad\quad \langle D@SD\rangle\mathtt{register!}\,[B\ \mathtt{rack}] \\
\qquad\qquad\quad \mid \mathtt{rack?}[] \rightarrow \mathbf{iflocal}\ \langle a\rangle\mathtt{pack!}\,[]\ \mathbf{then} \\
\qquad\qquad\qquad \mathtt{currentloc!}\,[S\ DS] \mid [\![P]\!]_b \\
\qquad\quad \mathbf{in} \\
\qquad\qquad \mathbf{let}\ b = [B\ S\ DS]\ \mathbf{in} \\
\qquad\qquad\quad \mathtt{pack?}[] \rightarrow \mathtt{currentloc!}\,[S\ DS] \mid [\![Q]\!]_a \\
\\
[\![\mathbf{migrate\ to}\ s \rightarrow P]\!]_a \\
\quad \overset{\text{def}}{=}\ \ \mathtt{currentloc?}[S\ DS] \rightarrow \\
\qquad\quad \mathbf{let}\ [U\ DU] = s\ \mathbf{in} \\
\qquad\qquad \mathbf{if}\ [S\ DS] = [U\ DU]\ \mathbf{then} \\
\qquad\qquad\quad \mathtt{currentloc!}\,[U\ DU] \\
\qquad\qquad \mathbf{else} \\
\qquad\qquad\quad \mathbf{new}\ \mathtt{mack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \\
\qquad\qquad\qquad \langle D@SD\rangle\mathtt{migrating!}\,[a\ \mathtt{mack}] \\
\qquad\qquad\qquad \mid \mathtt{mack?}[\mathtt{migrated}] \rightarrow \\
\qquad\qquad\qquad\quad \mathbf{migrate\ to}\ s \rightarrow \\
\qquad\qquad\qquad\qquad \mathbf{new}\ \mathtt{ack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \langle DU@U\rangle\mathtt{migrated!}\,[a\ [U\ DU]\ \mathtt{ack}] \\
\qquad\qquad\qquad\qquad\quad \mid \mathtt{ack?}[] \rightarrow \mathtt{currentloc!}\,s \mid [\![P]\!]_a \\
\\
[\![\mathbf{iflocal}\ \langle b\rangle c\,\mathtt{!}\,v\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!]_a \\
\quad \overset{\text{def}}{=}\ \ \mathbf{let}\ [B\ \_\ \_] = b\ \mathbf{in} \\
\qquad\qquad \mathbf{iflocal}\ \langle B\rangle c\,\mathtt{!}\,v\ \mathbf{then}\ [\![P]\!]_a\ \mathbf{else}\ [\![Q]\!]_a
\end{array}
$$

Figure A.2: Forwarding-pointer: the compositional encoding (selected clauses) where *map* is a map such that $map(a) = [s_j\ D_j]$ if $\Phi, \Delta \vdash a@s_j$.[1] The body of the daemon and selected clauses of the compositional translations are shown in Figures A.1 and A.2.

A.1 Sketch of Intermediate Language

In this section, we sketch an intermediate language that could be used for proving the correctness of the above encoding.

In this encoding, the `lock`- and `currentloc`-acquisition steps remain partial commitment steps, since they introduce internal choice. Acquisition of `Bstate`, the lock of an agent b at the daemon, is also a partial commitment step since there can be many message-forwarding requests to b (each of which first tries to acquire the lock `Bstate`). The syntax of the intermediate of this encoding may be as follows:

$$Sys \quad ::= \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$$

As in the intermediate language of the $\mathcal{C}$-encoding, Δ is the top-level declarations. However, the state of the daemons $\boldsymbol{D}$ is now a map, mapping each site s to a tuple $[\mathsf{mesgQ}\ f_s]$, where mesgQ is a queue of message-forwarding requests and f_s corresponds to the site map maintained by the daemon at s. The site map f_s is a partial function, mapping each agent a (that is or was at s) to a tuple $[S\ DS]$ where DS, located at S, is the daemon where messages to a should be forwarded to. The example below shows the state of two daemons, DS_1 and DS_2. The daemon DS_1 is at site S_1, and contains message queue mesgQ_1, similarly for DS_2. The site map of DS_1 indicates that an agent a_1 is at the same site as DS_1, and that messages for an agent a_2 should be forwarded to the daemon DS_2 at site S_2. The site map of DS_2 only have a record of a single agent, a_2, which is indicated to be at the same site as DS_2.

$$S_1 \mapsto [\mathsf{mesgQ}_1\ (a_1 \mapsto [S_1\ DS_1], a_2 \mapsto [S_2\ DS_2])], \quad S_2 \mapsto [\mathsf{mesgQ}_2\ (a_2 \mapsto [S_2\ DS_2])]$$

The agent state $\boldsymbol{A}$ is also a map, mapping each agent a to its state $[P\ \boldsymbol{E}]$. The pending state $\boldsymbol{E}$ of each agent, however, is more complicated than in the $\mathcal{C}$-encoding, since the $\mathcal{FP}$-encoding introduces many more partial-commitment steps to creation, migration and LI outputs. Two partially-committed steps are introduced for each creation, corresponding in the target level to the parent acquiring `currentloc`, and the registration request acquiring `lock` at the local daemon. $2 * n + 1$ steps are introduced for each LI message, where n is the

[1] In fact the map of each daemon may take different value, provided that the total view of the maps satisfies Eq. A.1.

number of forwarding pointers the message has to follow, corresponding in the target level to the sender acquiring `currentloc`, the message-forwarding request acquiring `lock` at the local daemon, and the request acquiring `Bstate` of the target agent — the latter two operations are repeated n times. Seven for each migration of an agent a, corresponding in the target level to a acquiring `currentloc`, then the `migrating`-request acquiring `lock` at the local daemon; such a request then acquires `Bstate` of a, a performs a migration, a (re)registers to the daemon at the new site — acquiring `lock` and possibly `Bstate` at such a site; and a leaves a forwarding pointer to the daemon of its previous site — acquiring `lock`). The rest of the additional τ-steps are housekeeping steps, which can be abstracted away in the intermediate language. The syntax of pending states is therefore as follows:

$$\boldsymbol{E} \quad ::= \quad \mathsf{FreeA}(s) \mid \mathsf{RegA}_{reg}(b\ Z\ s\ P\ Q) \mid \mathsf{MtingA}_{mig}(s\ P)$$

where reg and mig are integers ranging from 1 to 2 and to 4, respectively — such numbers denoting the partially committed step the agent is currently in w.r.t. registration, and migration.

The message-forwarding operation in this case is clearly more complex than that of the $\mathcal{C}$-encoding. We denote a message-forwarding request as $\mathsf{MesgA}_{mes}(b\ c\ v)$, with b being the target agent for the output $c\,!\,v$. When following a chain of pointers, this request may move from one daemon to another. The typing rule of this intermediate language must make sure that the chain of pointers induced by $\boldsymbol{D}$ eventually terminates at the correct destination, ie. if $\Phi, \Delta \vdash a@s$ then, for each site s', the following holds

$$\begin{aligned} \boldsymbol{D}(s')(a) = [s_1\ D_1] \quad \Rightarrow \quad & \exists \text{ distinct } s_2, \ldots, s_k\ .\ \boldsymbol{D}(s_k)(a) = [s\ D] \wedge \\ & \forall 1 \leq j < k\ .\ \boldsymbol{D}(s_j)(a) = [s_{j+1}\ D_{j+1}] \end{aligned} \tag{A.1}$$

The integer mes denotes the partially-committed state this request is currently in *at the daemon in which this request resides* and may take a value of 1 or 2.

The exact syntax, typing rules and the labelled transition rules for this new intermediate language remain future work.

Appendix B

Meta-Theoretic Results

In this section, we may refer to an arbitrary name binding in a located type context as $x : T@z$, where $@z$ ranges over *optional location annotations*, which are of the form $@s$ when attached to agent type and empty otherwise.

B.1 Structural Congruence and the Type System

Lemma B.1.1 (Type context permutation)
If $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Xi) = \emptyset$, $\mathsf{range}(\Delta) \cap \mathsf{dom}(\Xi) = \emptyset$ and $\mathsf{range}(\Xi) \cap \mathsf{dom}(\Delta) = \emptyset$ then $\Gamma, \Delta, \Xi, \Theta \equiv \Gamma, \Xi, \Delta, \Theta$.

Proof: An induction on the syntax of Δ and Ξ. ■

Lemma B.1.2 (Relocation preserves typing)
If $\Gamma \vdash LP$ then, for any relocators δ valid for Γ, $\Gamma\delta \vdash LP$.

Lemma B.1.3 (Typing and free names)

1. If $\Gamma \vdash T$ then $\mathsf{fv}(T) \subseteq \mathsf{dom}(\Gamma)$.

2. If $\Gamma \vdash e \in T$ then $\mathsf{fv}(e) \subseteq \mathsf{dom}(\Gamma)$.

3. If $\Gamma \vdash p \in T \triangleright \Delta$ then $\mathsf{fv}(p) = \mathsf{dom}(\Delta)$.

4. If $\Gamma \vdash_a P$ then $\mathsf{fv}(P) \cup \{a\} \subseteq \mathsf{dom}(\Gamma)$.

5. If $\Gamma \vdash LP$ then $\mathsf{fv}(LP) \subseteq \mathsf{dom}(\Gamma)$.

Proof: A routine induction on type derivation for each judgement form. The only interesting case seems to be for patterns. The proof of the lemma for the pattern case is given below.

Base Case (Pat-Var): In this case, $\Gamma \vdash x \in T \triangleright x : T$, we have: $\vdash \Gamma$ and $\Gamma \vdash T$, by (PAT-VAR). $\mathsf{fv}(p) = x$. $\mathsf{dom}(\Delta) = x$. Thus true for this case.

Base Case (Pat-Wild): In this case, $\Gamma \vdash \_ \in T \triangleright \bullet$, we have: $\vdash \Gamma$ and $\Gamma \vdash T$, by (PAT-WILD). $\mathsf{fv}(p) = \emptyset$. $\mathsf{dom}(\Delta) = \emptyset$. Thus true for this case.

Inductive Case (Pat-Exist): In this case, $\Gamma \vdash \{X\}\, p \in \{X\}\, T \triangleright X, \Delta$, we have: $\Gamma, X \vdash p \in T \triangleright \Delta$ and $X \notin \mathsf{dom}(\Gamma)$, by (PAT-EXIST). $\mathsf{fv}(\{X\}\, p) = \{X\} \cup \mathsf{fv}(p)$. $\mathsf{dom}(X, \Delta) = \{X\} \cup \mathsf{dom}(\Delta)$. By the induction hypothesis, $\mathsf{fv}(p) = \mathsf{dom}(\Delta)$. Thus true for this case.

Inductive Case (Pat-Tuple): In this case, $\Gamma \vdash p = [p_1 \ldots p_n] \in [T_1 \ldots T_n] \triangleright \Delta_1, \ldots, \Delta_n$, we have: $\Gamma \vdash p_i \in T_i \triangleright \Delta_i$ for all $1 \leq i \leq n$, by (PAT-TUPLE). $\mathsf{fv}(p) = \mathsf{fv}(p_1) \cup \ldots \cup \mathsf{fv}(p_n)$. $\mathsf{dom}(\Delta) = \mathsf{dom}(\Delta_1) \cup \ldots \cup \mathsf{dom}(\Delta_n)$. By the induction hypothesis, $\mathsf{fv}(p_i) = \mathsf{dom}(\Delta_i)$. Thus true for this case.

Therefore Lemma B.1.3(3) is proved by induction. ■

Lemma B.1.4
$\Gamma \vdash p \in T_2 \triangleright \Theta_2$ and $\Gamma \vdash T_1 \leq T_2$ then there exists Θ_1 such that $\Gamma \vdash \in p \vdash T_1 \triangleright \Theta_1$.

Proof: An induction on the derivation of $\Gamma \vdash p \in T_2 \triangleright \Theta_2$. ■

Lemma B.1.5
Given that $\Gamma \vdash p \in T_1 \triangleright \Theta_1$, $\Gamma \vdash p \in T_2 \triangleright \Theta_2$, and $\Gamma \vdash T_1 \leq T_2$, if $\sigma : \mathsf{fv}(p) \to \mathsf{dom}(\Gamma)$ is type-preserving w.r.t. Γ, Θ_1 then σ is type-preserving w.r.t. Γ, Θ_2.

Proof: An induction on the derivation of $\Gamma \vdash p \in T_1 \triangleright \Theta_1$. ■

Lemma B.1.6
Given that $\Gamma \vdash S$, we have:

- $\vdash \Gamma, X, \Xi$ implies $\vdash \Gamma, \{S/X\}\Xi$;
- $\Gamma, X, \Xi \vdash e \in T$ implies $\Gamma, \{S/X\}\Xi \vdash e \in \{S/X\}T$; and
- $\Gamma, X \vdash p \in T \triangleright \Xi$ implies $\Gamma \vdash p \in \{S/X\}T \triangleright \{S/X\}\Xi$.

Proof: Routine inductions on typing derivation of $\vdash (\Gamma, X, \Xi)$, of $(\Gamma, X, \Xi) \vdash e \in T$, and of $\Gamma, X \vdash p \in T \triangleright \Xi$. ■

Lemma B.1.7
Given that $X \notin \mathsf{dom}(\sigma)$, if $\sigma : \mathsf{dom}(\Xi) \to \mathsf{dom}(\Gamma)$ is type-preserving w.r.t. $\Gamma, \{T/X\}\Xi$ then $(\sigma + X \mapsto T)$ is type-preserving w.r.t. Γ, X, Ξ.

Proof: Supposing $x \in \mathsf{dom}(\Gamma, X, \Xi)$ and $\Gamma, X, \Xi \vdash x \in S$. By Lemma B.1.6, we have: $\Gamma, \{T/X\}\Xi \vdash x \in \{T/X\}S$. σ is type-preserving w.r.t. $\Gamma, \{T/X\}\Xi$ implies $\Gamma, \{T/X\}\Xi \vdash (\sigma + X \mapsto T)x \in (\sigma + X \mapsto T)S$. $\sigma : \mathsf{dom}(\Xi) \to \mathsf{dom}(\Gamma)$ implies $\Gamma \vdash (\sigma + X \mapsto T)x \in (\sigma + X \mapsto T)S$, by Lemma 3.7.2.

Similar derivation can be given for type variables in Γ, X, Ξ. Hence the lemma. ■

Lemma B.1.8 (Ground type context ensures matching is type-preserving (3.8.3))
Given that Γ is a ground located type context, if $\Gamma \vdash v \in S$ and $\Gamma \vdash p \in S \triangleright \Xi$ then $\mathsf{match}(p, v)$ is defined, and is a type-preserving substitution w.r.t. Γ, Ξ.

Proof: The proof of this result uses an induction on typing derivations of $\Gamma \vdash v \in S$. We demonstrate three interesting cases below.

Case (Var-Id): Supposing $\Gamma \vdash v \in S$ is derived using only the rule (Var-In). Γ is ground implies v is not a tuple or an existential package. This means that $\Gamma \vdash p \in S \triangleright \Xi$ must be derived by either (Pat-Var) or (Pat-Wild). Case (Pat-Wild) is trivial. Supposing $p = z$, we have $\Xi = z : S$, by (Pat-Var). In this case, $\mathsf{match}(p, v) = \{v/z\}$, which is clearly type-preserving w.r.t. Γ, Ξ. Thus the lemma is true for this case.

Case (Expr-Sub): Supposing (Expr-Sub) is used last in deriving $\Gamma \vdash v \in S$. There exists T such that $\Gamma \vdash v \in T$ and $\Gamma \vdash T \leq S$. $\Gamma \vdash p \in S \triangleright \Xi$ implies there exists Ξ' such that $\Gamma \vdash p \in T \triangleright \Xi'$, by Lemma B.1.4. By the induction hypothesis, we have $\mathsf{match}(p, v)$ is defined and is type-preserving w.r.t. Γ, Ξ'. By Lemma B.1.5, $\mathsf{match}(p, v)$ is defined and is type-preserving w.r.t. Γ, Ξ. Thus the lemma is true for this case.

Case (Expr-Exist): Supposing (Expr-Exist) is used last in deriving $\Gamma \vdash v \in S$. There exists X, S', T, v' such that, $v = \{\!|T|\!\}\, v'$, $S = \{\!|X|\!\}\, S'$, $\Gamma \vdash v' \in \{T/X\}S'$, $\Gamma, X \vdash S'$ and $\Gamma \vdash T$.

- There exists p', Ξ' such that $p = \{\!|X|\!\}\, p'$, $\Gamma, X \vdash p' \in S' \triangleright \Xi'$ and $\Xi = X, \Xi'$, by (Pat-Exist).
- $\{T/X\}$ is a type-preserving substitution w.r.t. Γ, X.
- $\Gamma \vdash p' \in \{T/X\}S' \triangleright \{T/X\}\Xi'$, by Lemma B.1.6, and Lemma 3.7.2.
- $\mathsf{match}(p', v')$ is defined and is type-preserving w.r.t. $\Gamma, \{T/X\}\Xi'$, by the induction hypothesis.

- $\mathsf{match}(p, v)$ is defined and is type-preserving w.r.t. Γ, X, Ξ', by Lemma B.1.7.

Thus the lemma is true for this case. ■

Lemma B.1.9 (Substitutions preserves typing)
Given a ground located type context Γ, if $\Gamma \vdash p \in T \triangleright \Theta$, $(\Gamma, \Theta) \vdash_a P$, $\Gamma, \Delta \vdash v \in T$, Δ is extensible, and $a \in \mathsf{dom}(\Gamma)$ then $\Gamma, \Delta \vdash_a \mathsf{match}(p, v)\ P$.

Proof: Assuming that p is fresh (ie. $\mathsf{dom}(p) = \mathsf{fv}(\Theta) \cap \mathsf{dom}(\Delta) = \emptyset$), and $\Gamma, \Theta \vdash_a P$, we have:

- $\Gamma, \rho\Theta \vdash_a \rho P$, by Lemma 3.8.1, where ρ is an injection such that $\rho : \mathsf{dom}(\Theta) \to \mathcal{X}/\mathsf{dom}(\Gamma, \Delta)$.
- $\Gamma, \Delta, \rho\Theta \vdash_a \rho P$, by Lemma 3.7.2.
- $\mathsf{match}(\rho p, v)$ is defined, and is type-preserving w.r.t. $\Gamma, \Delta, \rho\Theta$, Lemma 3.8.3. We observe that $\mathsf{match}(p, v) : \mathsf{dom}(\rho\Theta) \to \mathsf{dom}(\Gamma, \Delta)$.
- $\Gamma, \Delta, \rho\Theta \vdash_a \mathsf{match}(\rho p, v)(\rho P)$, by Lemma 3.8.2. Note that $\mathsf{match}(\rho p, v)a = a$, since $a \in \mathsf{dom}(\Gamma)$.
- $\Gamma, \Delta \vdash_a \mathsf{match}(\rho p, v)\rho P$, by Lemma 3.7.2.
- Applying the substitution ρ^{-1} to the previous result, we obtain $\Gamma, \Delta \vdash_a \mathsf{match}(p, v)P$, by Lemma 3.8.1.

Hence the lemma. ■

B.2 Operational Semantics

Lemma B.2.1 (Transitions preserve free names)

1. If $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ then either:
   (a) $\alpha = c\mathbf{?}v$ and $\{c\} \subseteq \mathsf{fv}(P)$ with $\mathsf{fv}(LP) \subseteq \mathsf{fv}(P) \cup \mathsf{fv}(v) \cup \{a\}$ and $\mathsf{fv}(v) \subseteq \mathsf{dom}(\Gamma, \Delta)$; or
   (b) $\mathsf{fv}(\alpha) \subseteq \mathsf{fv}(P)$ and $\mathsf{fv}(LP) \subseteq \mathsf{fv}(P) \cup \{a\} \cup \mathsf{dom}(\Delta)$.

   Moreover, $\mathsf{range}(\Delta) \cap \mathsf{dom}(\Gamma) = \emptyset$.
2. If $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then either:

(a) $\beta = @_a c\mathbf{?}v$ and $\{a, c\} \subseteq \mathsf{fv}(LP)$ with $\mathsf{fv}(LQ) \subseteq \mathsf{fv}(LP) \cup \mathsf{fv}(v)$ and $\mathsf{fv}(v) \subseteq \mathsf{dom}(\Gamma, \Delta)$; or

(b) $\mathsf{fv}(\beta) \subseteq \mathsf{fv}(LP)$ and $\mathsf{fv}(LQ) \subseteq \mathsf{fv}(LP) \cup \mathsf{dom}(\Delta)$.

Moreover, $\mathsf{range}(\Delta) \cap \mathsf{dom}(\Gamma) = \emptyset$.

Proof: Routine inductions on the derivation of $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$, and of $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$. ■

Lemma B.2.2 (Type context permutation preserves transition)
If $\Gamma \Vdash LP \xrightarrow[\Xi]{\beta} LQ$ and $\Delta \equiv \Gamma$ then $\Delta \Vdash LP \xrightarrow[\Xi]{\beta} LQ$.

Proof: A routine induction on the derivation of $\Gamma \Vdash LP \xrightarrow[\Xi]{\beta} LQ$. ■

Lemma B.2.3 (Transition analysis)

1. $\Gamma \Vdash_a P|Q \xrightarrow[\Delta]{\alpha} LP$ iff one of the following (or symmetric cases) holds.

   (a) There exists LP' such that $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP'$ and $LP \equiv LP'|@_a Q$.

   (b) There exists c, v, Δ, LP', LQ' such that $\Gamma \Vdash_a P \xrightarrow[\Delta]{c\mathbf{!}v} LP'$, $\Gamma \Vdash_a Q \xrightarrow[\Delta]{c\mathbf{?}v} LQ'$ and $LP \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ LP'|LQ'$.

2. $\Gamma \Vdash_a \mathbf{new}\ \Theta\ \mathbf{in}\ P \xrightarrow[\Delta]{\alpha} LP$ iff one of the following (or symmetric cases) holds.

   (a) There exists LP' and an injection $\sigma : \mathsf{dom}(\Theta) \to \mathcal{X}/\mathsf{dom}(\Gamma, \Delta)$ such that $\Gamma, \sigma\Theta \Vdash_a P \xrightarrow[\Delta]{\alpha} LP'$ and $LP \equiv \mathbf{new}\ \sigma\Theta\ \mathbf{in}\ LP'$.

   (b) There exists $c, v, \Theta_1, \Theta_2, LP'$ such that $\alpha = c\mathbf{!}v$, $\Theta \equiv \Theta_1, \Theta_2$, $\mathsf{dom}(\Theta) \cap \mathsf{fv}(v) = \mathsf{dom}(\Theta_1)$, $\Gamma, \Theta \Vdash_a P \xrightarrow[\Delta/\Theta_1]{@_a c\mathbf{!}v} LP'$ and $LP \equiv \mathbf{new}\ \Theta_2\ \mathbf{in}\ LP'$.

3. $\Gamma \Vdash LP|LQ \xrightarrow[\Delta]{\beta} LR$ iff one of the following (or symmetric cases) holds.

   (a) There exists LP' such that $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ and $LR \equiv LP'|LQ$.

   (b) There exists $a, c, v, \Delta, LP', LQ'$ such that $\Gamma \Vdash LP \xrightarrow[\Delta]{@_a c\mathbf{!}v} LP'$, $\Gamma \Vdash LQ \xrightarrow[\Delta]{@_a c\mathbf{?}v} LQ'$ and $LR \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ LP'|LQ'$.

4. $\Gamma \Vdash \mathbf{new}\ \Theta\ \mathbf{in}\ LP \xrightarrow[\Delta]{\beta} LQ$ iff one of the following (or symmetric cases) holds.

   (a) There exists LP' and an injection $\sigma : \mathsf{dom}(\Theta) \to \mathcal{X}/\mathsf{dom}(\Gamma, \Delta)$ such that $\Gamma, \sigma\Theta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ and $LQ \equiv \mathbf{new}\ \sigma\Theta\ \mathbf{in}\ LP'$.

(b) $\beta = \tau$ and there exists $a \in \mathsf{dom}(\Theta)$, s such that $\Gamma, \Theta \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s} LP'$ and $LQ \equiv \mathbf{new}\ \Theta \oplus a \mapsto s\ \mathbf{in}\ LP'$.

(c) There exists $a, c, v, \Theta_1, \Theta_2, LP'$ such that $\beta = @_a c\mathbf{!}v$, $\Theta \equiv \Theta_1, \Theta_2$, $\mathsf{dom}(\Theta) \cap \mathsf{fv}(v) = \mathsf{dom}(\Theta_1)$, $\Gamma, \Theta \Vdash LP \xrightarrow[\Delta/\Theta_1]{@_a c\mathbf{!}v} LP'$ and $LQ \equiv \mathbf{new}\ \Theta_2\ \mathbf{in}\ LP'$.

Proof: Routine inductions on the transition derivation. ■

Lemma B.2.4 (Injection preserves transition (4.5.7))
Given that $\rho : \mathsf{dom}(\Gamma) \to \mathcal{X}$ is injective and $\sigma : \mathsf{dom}(\Delta) \to \mathcal{X}$ is injective and $\mathsf{range}(\rho) \cap \mathsf{range}(\sigma) = \emptyset$, we have:

1. if $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ then $\rho\Gamma \Vdash_{\rho a} \rho P \xrightarrow[\sigma\Delta]{(\rho+\sigma)\alpha} (\rho + \sigma)LP$; and
2. if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\rho\Gamma \Vdash \rho LP \xrightarrow[\sigma\Delta]{(\rho+\sigma)\beta} (\rho + \sigma)LQ$.

Proof: Routine inductions on the transition derivation. We shall demonstrate a few interesting cases here.

Base Case (Lts-In): Supposing $\Gamma \Vdash_a c\mathbf{?}p \to P \xrightarrow[\Delta]{c\mathbf{?}v} @_a\mathsf{match}(p, v)P$, and (Lts-In) is used last for deriving this transition. We may derive the following.

- By (Lts-In), we have $\Gamma \vdash c \in \mathbf{\hat{}^{r}}T$, $(\Gamma, \Delta) \vdash v \in T$, $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v)$ and Δ extensible.
- By Lemma 3.8.1 (Type Inj), we have $\rho\Gamma \vdash \rho c \in \mathbf{\hat{}^{r}}(\rho T)$, $(\rho\Gamma, \sigma\Delta) \vdash (\rho + \sigma)v \in \rho T$. Moreover, $\mathsf{dom}(\sigma\Delta) \subseteq \mathsf{fv}((\rho + \sigma)v)$ and $\sigma\Delta$ is extensible.
- Pick an injection $\sigma' : \mathsf{fv}(p) \to \mathcal{X}/(\mathsf{dom}(\Gamma, \Delta) \cup \mathsf{range}(\sigma + \rho))$, by (Lts-In), we have:

$$\rho\Gamma \Vdash_{\rho a} \rho c\mathbf{?}(\sigma' p) \to (\rho + \sigma')P \xrightarrow[\sigma\Delta]{(\rho+\sigma)(c\mathbf{?}v)} @_{\rho a}\mathsf{match}(\sigma' p, (\rho + \sigma)v)(\rho + \sigma')P$$

- $\rho c\mathbf{?}(p\sigma') \to (\rho + \sigma')P \overset{\alpha}{=} \rho c\mathbf{?}p \to \rho P = \rho(c\mathbf{?}p \to P)$.
- $\mathsf{match}(\sigma' p, (\rho + \sigma)v)((\rho + \sigma')P) \overset{\alpha}{=} \mathsf{match}(p, (\rho + \sigma)v)(\rho P) = (\rho + \sigma)(\mathsf{match}(p, v)P)$.

Thus true for this case.

Inductive Case (Lts-Comm): Supposing (Lts-Comm) is used last in deriving

$$\Gamma \Vdash LP|LQ \xrightarrow{\tau} \mathbf{new}\ \Delta\ \mathbf{in}\ (LP'|LQ')$$

In this case, σ is an empty substitution. We may derive the following.

- $\Gamma \Vdash LP \xrightarrow[\Delta]{@_a c\mathbf{!}v} LP'$ and $\Gamma \Vdash LQ \xrightarrow[\Delta]{@_a c\mathbf{?}v} LQ'$, for some a, c, v, by (Lts-Comm).

- Pick an injection $\sigma' : \mathsf{dom}(\Delta) \to \mathcal{X}/(\mathsf{dom}(\Gamma) \cup \mathsf{range}(\rho))$, by the induction hypothesis, we have:

$$\rho\Gamma \Vdash \rho LP \xrightarrow[\sigma'\Delta]{(\rho+\sigma')(@_a c!v)} (\rho+\sigma')LP' \quad \text{and}$$
$$\rho\Gamma \Vdash \rho LQ \xrightarrow[\sigma'\Delta]{(\rho+\sigma')(@_a c?v)} (\rho+\sigma')LQ'$$

- $\rho\Gamma \Vdash (\rho LP)|(\rho LQ) \xrightarrow{\tau} \mathbf{new}\ \sigma'\Delta\ \mathbf{in}\ ((\rho+\sigma')LP')|((\rho+\sigma')LQ')$, by (Lts-Comm).
- $\mathbf{new}\ \sigma'\Delta\ \mathbf{in}\ ((\rho+\sigma')LP')|((\rho+\sigma')LQ') \stackrel{\alpha}{=} \rho(\mathbf{new}\ \Delta\ \mathbf{in}\ (LP'|LQ'))$ and $(\rho LP)|(\rho LQ) = \rho(LP|LQ)$.

Thus true for this case.

Inductive Case (Lts-New): Supposing (Lts-New) is used last in deriving

$$\Gamma \Vdash \mathbf{new}\ x : T@z\ \mathbf{in}\ LP \xrightarrow[\Delta]{\beta} LQ$$

We may derive the following.

- $\Gamma, x : T@z \Vdash LP \xrightarrow[\Delta]{\beta} LQ'$, $LQ = \mathbf{new}\ x : T@z\ \mathbf{in}\ LQ'$ and $x \notin \mathsf{fv}(\beta)$, by (Lts-New).
- Pick a substitution $\sigma' : \{x\} \to \mathcal{X}/(\mathsf{dom}(\Gamma,\Delta) \cup \mathsf{range}(\rho,\sigma))$, by the induction hypothesis, we have:

$$(\rho+\sigma')(\Gamma, x : T@z) \Vdash (\rho+\sigma')LP \xrightarrow[\sigma\Delta]{(\rho+\sigma+\sigma')\beta} (\rho+\sigma+\sigma')LQ'$$

- By (Lts-New), we have:

$$\begin{aligned}\rho\Gamma \Vdash\ & \mathbf{new}\ (\rho+\sigma')(x : T@z)\ \mathbf{in}\ (\rho+\sigma')LP \\ & \xrightarrow[\sigma\Delta]{(\rho+\sigma+\sigma')\beta} \mathbf{new}\ (\rho+\sigma')(x : T@z)\ \mathbf{in}\ (\rho+\sigma+\sigma')LQ'\end{aligned}$$

- $\mathbf{new}\ (\rho+\sigma')(x : T@z)\ \mathbf{in}\ (\rho+\sigma')LP \stackrel{\alpha}{=} \rho(\mathbf{new}\ x : T@z\ \mathbf{in}\ LP)$
- $\mathbf{new}\ (\rho+\sigma')(x : T@z)\ \mathbf{in}\ (\rho+\sigma+\sigma')LQ' \stackrel{\alpha}{=} (\rho+\sigma)(\mathbf{new}\ x : T@z\ \mathbf{in}\ LQ')$.

Thus true for this case.

Inductive Case (Lts-Open): Supposing (Lts-Open) is used last in deriving

$$\Gamma \Vdash \mathbf{new}\ x : T@z\ \mathbf{in}\ LP \xrightarrow[\Delta, x:T@z]{@_a c!v} LQ$$

We may derive the following.

- $\Gamma, x : T@z \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$ and $x \in \mathsf{fv}(v)$, by (Lts-Open).
- Let $\rho' = \rho + \sigma''$, where $\sigma'' = x \mapsto \sigma(x)$ and $\sigma' = \sigma/\{x\}$, by the induction hypothesis, we have:
$$\rho'(\Gamma, x : T@z) \Vdash \rho' LP \xrightarrow[\sigma'\Delta]{(\rho'+\sigma')(@_a c!v)} (\rho' + \sigma')LQ'$$
- $\rho\Gamma' \Vdash \mathbf{new}\ \rho'(x : T@z)\ \mathbf{in}\ \rho' LP \xrightarrow[\sigma'\Delta, \rho'(x:T@z)]{(\rho'+\sigma')(@_a c!v)} (\rho' + \sigma')LQ'$, by (Lts-Open).
- $\rho'\Gamma = \rho\Gamma$, $\mathbf{new}\ \rho'(x : T@z)\ \mathbf{in}\ \rho' LP \stackrel{\alpha}{=} \rho(\mathbf{new}\ x : T@z\ \mathbf{in}\ LP)$ and $(\rho' + \sigma') = (\rho + \sigma)$.

Thus true for this case.

■

Lemma B.2.5 (Weakening and strengthening Lts (unary name))
Given that $\vdash_{\mathsf{L}} \Gamma, x : T@z$,

1. if $\Gamma \vdash_a P$ and $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\alpha)$ then
$$\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP \ \Leftrightarrow\ (\Gamma, x : T@z) \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP$$
2. if $\Gamma \vdash LP$ and $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$ then
$$\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ \ \Leftrightarrow\ (\Gamma, x : T@z) \Vdash LP \xrightarrow[\Delta]{\beta} LQ$$

Proof: We prove this by an induction on the transition derivation. We shall only demonstrate interesting cases.

Base Case (Lts-L-Rep): From left-to-right direction, assuming $\vdash \Gamma, x : T@z$,
$$\Gamma \Vdash_a\ {*}c?p \to P \xrightarrow[\Delta]{c?v} @_a(\mathsf{match}(v, p)P \mid {*}c?p \to P)$$
and $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$, we may derive the following.

- $\Gamma \vdash c \in \hat{}^{\mathtt{r}}T$, $(\Gamma, \Delta) \vdash v \in T$ and $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v)$, by (Lts-L-Rep).
- Since $\vdash \Gamma, x : T@z$, applying Lemma 3.7.2 (Type SW), we have: $\Gamma, x : T@z \vdash c \in \hat{}^{\mathtt{r}}T$. Moreover, $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$ implies $\vdash \Gamma, \Delta, x : T@z$, by (LC-Var). Applying Lemma 3.7.2 (Type SW), we have: $\Gamma, \Delta, x : T@z \vdash v \in T$.
- $z \in \mathsf{dom}(\Gamma)$ implies $z \notin \mathsf{dom}(\Delta)$. Moreover, $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$, applying B.1.1, we have: $\Gamma, \Delta, x : T@z \equiv \Gamma, x : T@z, \Delta$.
- $\Gamma, x : T@z, \Delta \vdash v \in T$, by Lemma 3.7.1 (Type Perm).

- $\Gamma, x : T@z \Vdash_a\ \ast c?p \to P \xrightarrow[\Delta]{c?v} @_a\mathsf{match}(p,v)P \mid \ast c?p \to P$, by (LTS-L-REP).

From right-to-left direction, assuming $\Gamma \vdash_a \ast c?p \to P$,

$$\Gamma, x : T@z \Vdash_a\ \ast c?p \to P \xrightarrow[\Delta]{c?v} @_a(\mathsf{match}(v,p)P \mid \ast c?p \to P)$$

and $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$, we may derive the following.

- $\Gamma, x : T@z \vdash c \in \hat{}^{\mathbf{r}}T$, $(\Gamma, x : T@z, \Delta) \vdash v \in T$ and $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v)$, by (LTS-L-REP).
- Since $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$ and $z \in \mathsf{dom}(\Gamma)$ (ie. $z \notin \mathsf{dom}(\Delta)$), applying Lemma B.1.1, we have: $\Gamma, x : T@z, \Delta \equiv \Gamma, \Delta, x : T@z$. This implies $\Gamma, \Delta, x : T@z \vdash v \in T$, by Lemma 3.7.1 (Type Perm).
- $x \notin \mathsf{fv}(c,v)$, implies $\Gamma \vdash c \in \hat{}^{\mathbf{r}}T$ and $\Gamma, \Delta \vdash v \in T$, by Lemma 3.7.2 (Type SW).
- $\Gamma \Vdash_a\ \ast c?p \to P \xrightarrow[\Delta]{c?v} @_a\mathsf{match}(p,v)P \mid \ast c?p \to P$, by (LTS-L-REP).

Thus true for this case.

Base Case (Lts-Local): From left-to-right direction, supposing $\vdash \Gamma, x : T@z$, $\Gamma \Vdash @_aP \xrightarrow[\Delta]{\beta} LP$ and $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$, we may derive the following.

- $\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP$ where $\beta = \tau = \alpha$ or $\beta = @_a\alpha$, by (LTS-LOCAL).
- $\Gamma \vdash @_aP$ implies $x \neq a$. This means $\Gamma, x : T@z \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP$, by Lemma B.2.5 (basic process).
- $\Gamma, x : T@z \Vdash @_aP \xrightarrow[\Delta]{\beta} LP$, by (LTS-LOCAL).

The right-to-left direction is similar. Thus true for this case.

Inductive Case (Lts-Open): From left-to-right direction, supposing $\vdash \Gamma, x : T@z$,

$$\Gamma \Vdash \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LP \xrightarrow[\Delta, y:T'@z']{@_a c!v} LQ$$

$x \notin \mathsf{fv}(\Delta, y : T'@z') \cup \mathsf{fv}(@_a c!v)$ and (LTS-OPEN) is used last for deriving this, we may derive the following.

- $\Gamma, y : T'@z' \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$ and $y \in \mathsf{fv}(v)$, by (LTS-OPEN).
- $x \notin \mathsf{fv}(\Delta, y : T'@z') \cup \mathsf{fv}(@_a c!v)$ implies $x \notin \mathsf{fv}(\Delta)$. $\Gamma \vdash \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LP$ implies $\Gamma, y : T'@z' \vdash LP$, by (NEWCHAN/AGENT); moreover, $\vdash \Gamma, x : T@z$ implies $\vdash \Gamma, y : T'@z', x : T@z$.
- $\Gamma, y : T'@z', x : T@z \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$, by the induction hypothesis.

- $x \notin \mathsf{fv}(y : T'@z') \cup \mathsf{fv}(@_a c!v)$ and $y \neq x, z$ (as $z \in \mathsf{dom}(\Gamma)$), $\Gamma, y : T'@z', x : T@z \equiv \Gamma, x : T@z, y : T'@z'$, by Lemma B.1.1.
- $\Gamma, x : T@z, y : T'@z' \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$, by Lemma B.2.2 (Lts Perm).
- $\Gamma, x : T@z \Vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP \xrightarrow[\Delta, y:T'@z']{@_a c!v} LQ$, by (Lts-L-Open)

From right-to-left direction, supposing $\Gamma \vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP$,

$$\Gamma, x : T@z \Vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP \xrightarrow[\Delta, y:T'@z']{@_a c!v} LQ$$

$x \notin \mathsf{fv}(\Delta, y : T'@z') \cup \mathsf{fv}(@_a c!v)$ and (Lts-Open) is used last for deriving this, we may derive the following.

- $\Gamma, x : T@z, y : T'@z' \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$ and $y \in \mathsf{fv}(v)$, by (Lts-Open).
- $x \notin \mathsf{fv}(\Delta, y : T'@z') \cup \mathsf{fv}(@_a c!v)$ and $y \neq x, z$ implies $\Gamma, y : T'@z', x : T@z \equiv \Gamma, x : T@z, y : T'@z'$, by Lemma B.1.1.
- $\Gamma, y : T'@z', x : T@z \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$, by Lemma B.2.2 (Lts Perm).
- $\Gamma \vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP$ implies $\Gamma, y : T'@z' \vdash LP$, by (NewChan/Agent).
- $\Gamma, y : T'@z' \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$, by the induction hypothesis.
- $\Gamma \Vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP \xrightarrow[\Delta, y:T'@z']{@_a c!v} LQ$, by (Lts-L-Open)

Thus true for this case.

Inductive Case (Lts-New): From left-to-right direction, supposing $\vdash \Gamma, x : T@z$,

$$\Gamma \Vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP \xrightarrow[\Delta]{\beta} \texttt{new}\ y : T'@z'\ \texttt{in}\ LQ$$

$x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$ and (Lts-New) is used last for deriving this, we may derive the following.

- Pick a $\sigma : \{y\} \to \mathcal{X}/\mathsf{dom}(\Gamma, \Delta, x)$, we have $\texttt{new}\ y : T'@z'\ \texttt{in}\ LP \stackrel{\alpha}{=} \texttt{new}\ \hat{y} : T'@z'\ \texttt{in}\ \hat{LP}$, where $\hat{y} = \sigma y$ etc.
- $\Gamma, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta]{\hat{\beta}} \hat{LQ}$, by (Lts-New).
- $\vdash \Gamma, x : T@z$ and $x \neq \hat{y}$ implies $\vdash \Gamma, \hat{y} : T'@z', x : T@z$, by (L-C-Var).
- $\Gamma, \hat{y} : T'@z', x : T@z \Vdash \hat{LP} \xrightarrow[\Delta]{\hat{\beta}} \hat{LQ}$, by the induction hypothesis.
- Since $x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$ and $\hat{y} \neq x, z$, we have $\Gamma, \hat{y} : T'@z', x : T@z \equiv \Gamma, x : T@z, \hat{y} : T'@z'$, by Lemma B.1.1.

- $\Gamma, x : T@z, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta]{\hat{\beta}} \hat{LQ}$, by Lemma B.2.2 (Lts Perm).
- $\Gamma, x : T@z \Vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP \xrightarrow[\Delta]{\beta} \texttt{new}\ y : T'@z'\ \texttt{in}\ LQ$, by (Lts-L-New) and alpha-conversion.

From right-to-left direction, supposing $\Gamma \vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP$ and

$$\Gamma, x : T@z \Vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP \xrightarrow[\Delta]{\beta} \texttt{new}\ y : T'@z'\ \texttt{in}\ LQ$$

$x \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$ and (Lts-New) is used last for deriving this, we may derive the following.

- Pick a $\sigma : \{y\} \to \mathcal{X}/\mathsf{dom}(\Gamma, \Delta, x)$, we have $\texttt{new}\ y : T'@z'\ \texttt{in}\ LP \stackrel{\alpha}{=} \texttt{new}\ \hat{y} : T'@z'\ \texttt{in}\ \hat{LP}$, where $\hat{y} = \sigma y$ etc.
- $\Gamma, x : T@z, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta]{\beta} \hat{LQ}$, by (Lts-New) and alpha-conversion.
- $x \notin \mathsf{fv}(\texttt{new}\ \hat{y} : T'@z'\ \texttt{in}\ LP) \cup \mathsf{fv}(\beta)$ implies $x \neq z'$, $z \in \mathsf{dom}(\Gamma)$ implies $\hat{y} \neq z$. Applying Lemma B.1.1, $\Gamma, x : T@z, \hat{y} : T'@z' \equiv \Gamma, \hat{y} : T'@z', x : T@z$.
- $\Gamma, \hat{y} : T'@z', x : T@z \Vdash \hat{LP} \xrightarrow[\Delta]{\beta} \hat{LQ}$, by Lemma B.2.2 (Lts Perm).
- $\Gamma \vdash \texttt{new}\ \hat{y} : T'@z'\ \texttt{in}\ \hat{LP}$ implies $\Gamma, \hat{y} : T'@z' \vdash \hat{LP}$, by (NewChan/Agent). This means $\Gamma, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta]{\beta} \hat{LQ}$, by the induction hypothesis.
- $\Gamma \Vdash \texttt{new}\ y : T'@z'\ \texttt{in}\ LP \xrightarrow[\Delta]{\beta} \texttt{new}\ y : T'@z'\ \texttt{in}\ LQ$, by (Lts-L-New) and alpha-conversion.

Thus true for this case.

Inductive Case (Lts-Bound-Mig): From left-to-right direction, supposing $\vdash \Gamma, x : T@z$,

$$\Gamma \Vdash \texttt{new}\ a : \texttt{Agent}^{\texttt{m}}@s\ \texttt{in}\ LP \xrightarrow{\tau} \texttt{new}\ a : \texttt{Agent}^{\texttt{m}}@s'\ \texttt{in}\ LQ$$

and (Lts-Bound-Mig) is used last for deriving this, we may derive the following.

- $\Gamma, a : \texttt{Agent}^{\texttt{m}}@s \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s'} LQ$, by (Lts-Bound-Mig).
- Pick a $\sigma : \{a\} \to \mathcal{X}/\mathsf{dom}(\Gamma, x)$, we have: $\Gamma, \hat{a} : \texttt{Agent}^{\texttt{m}}@s \Vdash \hat{LP} \xrightarrow{@_{\hat{a}} \mathsf{migrate\ to}\ s'} \hat{LQ}$ where $\hat{a} = \sigma a$ etc, by Lemma 4.5.7 (Lts Inj).
- $\vdash \Gamma, x : T@z$ and $x \neq \hat{a}$ implies $\vdash \Gamma, \hat{a} : T'@z', x : T@z$, by (L-C-Var).
- $\Gamma, \hat{a} : \texttt{Agent}^{\texttt{m}}@s, x : T@z \Vdash \hat{LP} \xrightarrow{@_{\hat{a}} \mathsf{migrate\ to}\ s'} \hat{LQ}$, by the induction hypothesis.
- $\vdash \Gamma, x : T@z$ implies $z \in \mathsf{dom}(\Gamma)$ (ie. $\hat{a} \neq x, z$). $\Gamma \vdash \texttt{new}\ a : \texttt{Agent}^{\texttt{m}}@s\ \texttt{in}\ LP$ implies $s \in \mathsf{dom}(\Gamma)$ (ie. $x \neq \hat{a}, s$). Hence $\Gamma, \hat{a} : \texttt{Agent}^{\texttt{m}}@s, x : T@z \equiv \Gamma, x : T@z, \hat{a} : \texttt{Agent}^{\texttt{m}}@s$, by Lemma B.1.1.

- $\Gamma, x : T@z, \hat{a} : \mathtt{Agent^m}@s \Vdash \hat{LP} \xrightarrow{@_{\hat{a}}\mathsf{migrate\ to}\ s'} \hat{LQ}$, by Lemma B.2.2 (Lts Perm).
- $\Gamma, x : T@z \Vdash \mathbf{new}\ a : \mathtt{Agent^m}@s\ \mathbf{in}\ LP \xrightarrow{\tau} \mathbf{new}\ a : \mathtt{Agent^m}@s'\ \mathbf{in}\ LQ$, by (Lts-L-New) and alpha-conversion.

From right-to-left direction, supposing $\Gamma \vdash \mathbf{new}\ a : \mathtt{Agent^m}@s\ \mathbf{in}\ LP$ and

$$\Gamma, x : T@z \Vdash \mathbf{new}\ a : \mathtt{Agent^m}@s\ \mathbf{in}\ LP \xrightarrow{\tau} LQ$$

and (Lts-Bound-Mig) is used last for deriving this, we may derive the following.

- $\Gamma, x : T@z, a : \mathtt{Agent^m}@s \Vdash LP \xrightarrow{@_a\mathsf{migrate\ to}\ s'} LQ$, by (Lts-Bound-Mig).
- Since $\Gamma \vdash \mathbf{new}\ a : \mathtt{Agent^m}@s\ \mathbf{in}\ LP$ implies $s \in \mathsf{dom}(\Gamma)$ (ie. $x \neq s$). a is of an agent type hence $a \neq z$. This means $\Gamma, x : T@z, a : \mathtt{Agent^m}@s \equiv \Gamma, a : \mathtt{Agent^m}@s, x : T@z$, by Lemma B.1.1.
- $\Gamma, a : \mathtt{Agent^m}@s, x : T@z \Vdash LP \xrightarrow[\Delta]{@_a\mathsf{migrate\ to}\ s'} LQ$, by Lemma B.2.2 (Lts Perm).
- $\Gamma \vdash \mathbf{new}\ a : \mathtt{Agent^m}@s\ \mathbf{in}\ LP$ implies $\Gamma, a : \mathtt{Agent^m}@s \vdash LP$, by (NewAgent).
- $\Gamma, a : \mathtt{Agent^m}@s \Vdash LP \xrightarrow[\Delta]{@_a\mathsf{migrate\ to}\ s'} LQ$, by the induction hypothesis.
- $\Gamma \Vdash \mathbf{new}\ a : \mathtt{Agent^m}@s\ \mathbf{in}\ LP \xrightarrow{\tau} \mathbf{new}\ a : \mathtt{Agent^m}@s'\ \mathbf{in}\ LQ$, by (Lts-L-New)

Thus true for this case.

Therefore the Lemma is proved by induction. ■

Lemma B.2.6 (Weakening and strengthening Lts (unary type variable))
Given that $\vdash_{\mathsf{L}} \Gamma, X$,

1. if $\Gamma \vdash_a P$ and $X \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\alpha)$ then

$$\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP \ \Leftrightarrow\ (\Gamma, X) \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP$$

2. if $\Gamma \vdash LP$ and $X \notin \mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)$ then

$$\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ \ \Leftrightarrow\ (\Gamma, X) \Vdash LP \xrightarrow[\Delta]{\beta} LQ$$

Proof: Similar to the proof of Lemma B.2.5. ■

Lemma B.2.7 (Weakening and strengthening transition (4.5.5))
Given that $\vdash_{\mathsf{L}} \Gamma, \Theta$,

1. $\Gamma \vdash_a P$ and $\mathsf{dom}(\Theta) \cap (\mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)) = \emptyset$ implies

$$\Gamma \Vdash_a \ P \xrightarrow[\Delta]{\alpha} LP \ \Leftrightarrow \ (\Gamma, \Theta) \Vdash_a \ P \xrightarrow[\Delta]{\alpha} LP$$

2. $\Gamma \vdash LP$ and $\mathsf{dom}(\Theta) \cap (\mathsf{fv}(\Delta) \cup \mathsf{fv}(\beta)) = \emptyset$ implies

$$\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ \ \Leftrightarrow \ (\Gamma, \Theta) \Vdash LP \xrightarrow[\Delta]{\beta} LQ$$

Proof: An induction on the syntax of Θ, applying Lemma B.2.5 and Lemma B.2.6. ■

Lemma B.2.8 ((Unary) input shifting)
Given that $\vdash_{\mathsf{L}} \Gamma, x : T@z$, T is an extensible type and $a, c \neq x \in \mathsf{fv}(v)$, we have:

1. $\Gamma \vdash_a P$ implies $\Gamma \Vdash_a \ P \xrightarrow[\Delta, x:T@z]{c?v} LP \ \Leftrightarrow \ \Gamma, x : T@z \Vdash_a \ P \xrightarrow[\Delta]{c?v} LP$; and
2. $\Gamma \vdash LP$ implies $\Gamma \Vdash LP \xrightarrow[\Delta, x:T@z]{@_a c?v} LQ \ \Leftrightarrow \ \Gamma, x : T@z \Vdash LP \xrightarrow[\Delta]{@_a c?v} LQ$.

Proof: We prove these by inductions on the transition derivation. Since the transition undergoes an input action, the rules used for deriving this must be a combination of (LTS-L-(REP)IN, LTS-(L)-NEW, LTS-(L)-PRL). We only demonstrate (LTS-L-IN) and (LTS-NEW).

Base Case :(LTS-L-IN) From left-to-right direction, supposing $\vdash \Gamma, x : T@z$,

$$\Gamma \Vdash_a \ {*c?p} \rightarrow P \xrightarrow[\Delta, x:T@z]{c?v} LP$$

and (LTS-L-IN) is used last for deriving this. We may derive the following.

- By (LTS-L-IN), we have: $\Gamma \vdash c \in {}^{\wedge\mathtt{r}}T$, $(\Gamma, \Delta, x : T@z) \vdash v \in T$, $(\Delta, x : T@z)$ is extensible and $\mathsf{dom}(\Delta, x : T@z) \subseteq \mathsf{fv}(v)$.
- $\Gamma, x : T@z \vdash c \in {}^{\wedge\mathtt{r}}T$, by Lemma 3.7.2 (Type SW).
- $x \notin \mathsf{dom}(\Delta)$; moreover as $\Delta, x : T@z$ is extensible, $x \notin \mathsf{range}(\Delta)$ and $z \notin \mathsf{dom}(\Delta)$. This means $\Gamma, \Delta, x : T@z \equiv \Gamma, x : T@z, \Delta$, by Lemma B.1.1.
- $\Gamma, x : T@z, \Delta \vdash v \in T$, by Lemma 3.7.1 (Type Perm).
- $\Gamma, x : T@z \Vdash_a \ {*c?p} \rightarrow P \xrightarrow[\Delta]{c?v} LP$, by (LTS-L-IN).

From right-to-left direction, supposing $\Gamma \vdash_a {*c?p} \rightarrow P$,

$$\Gamma, x : T@z \Vdash_a \ {*c?p} \rightarrow P \xrightarrow[\Delta]{c?v} LP$$

T is extensible and $x \neq a$ with $x \in \mathsf{fv}(v)$, we may derive the following.

- By (LTS-L-IN), we have: $\Gamma, x : T@z \vdash c \in \hat{}^{\mathtt{r}}T$, $(\Gamma, x : T@z, \Delta) \vdash v \in T$, Δ is extensible and $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v)$.
- $\Gamma \vdash_a *c?p \to P$ implies $c \in \mathsf{dom}(\Gamma)$. Hence $\Gamma \vdash c \in \hat{}^{\mathtt{r}}T$, by Lemma 3.7.2 (Type SW).
- $x \notin \mathsf{dom}(\Delta)$; moreover as $\Delta, x : T@z$ is extensible, $x \notin \mathsf{range}(\Delta)$ and $z \notin \mathsf{dom}(\Delta)$. This means $\Gamma, \Delta, x : T@z \equiv \Gamma, x : T@z, \Delta$, by Lemma B.1.1.
- $\Gamma, \Delta, x : T@z \vdash v \in T$, by Lemma 3.7.1 (Type Perm).
- $\Gamma \Vdash_a *c?p \to P \xrightarrow[\Delta, x:T@z]{c?v} LP$, by (LTS-L-IN).

Thus true for the base case.

Inductive Case (Lts-New): From left-to-right direction, supposing $\vdash \Gamma, x : T@z$,

$$\Gamma \Vdash \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LP \xrightarrow[\Delta, x:T@z]{c?v} \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LQ$$

and (LTS-NEW) is used last for deriving this, we may derive the following.

- Pick a $\sigma : y \to \mathcal{X}/\mathsf{dom}(\Gamma, \Delta, x : T@z)$, we have $LP \stackrel{\alpha}{=} \mathbf{new}\ \hat{y} : T'@z'\ \mathbf{in}\ \hat{LP}$, where $\hat{y} = \sigma y$ etc.
- $\Gamma, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta, x:T@z]{c?v} \hat{LQ}$, by (LTS-NEW).
- $\Gamma, \hat{y} : T'@z', x : T@z \Vdash \hat{LP} \xrightarrow[\Delta]{c?v} \hat{LQ}$, by the induction hypothesis.
- $x \neq \hat{y}$; moreover, as $x, \hat{y}$ are of extensible types $z, z', x, \hat{y}$ are distinct. This means $\Gamma, \hat{y} : T'@z', x : T@z \equiv \Gamma, x : T@z, \hat{y} : T'@z'$, by Lemma B.1.1.
- $\Gamma, x : T@z, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta]{c?v} \hat{LQ}$, by Lemma B.2.2 (Lts Perm).
- $\Gamma, x : T@z \Vdash \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LP \xrightarrow[\Delta]{c?v} \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LQ$, by (LTS-NEW) and alpha-conversion.

From right-to-left direction, supposing $\Gamma \vdash \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LP$,

$$\Gamma, x : T@z \Vdash \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LP \xrightarrow[\Delta]{c?v} \mathbf{new}\ y : T'@z'\ \mathbf{in}\ LQ$$

T is extensible and $x \neq a$ with $x \in \mathsf{fv}(v)$, and (LTS-NEW) is used last for deriving this, we may derive the following.

- Pick a $\sigma : y \to \mathcal{X}/\mathsf{dom}(\Gamma, \Delta, x : T@z)$, we have $LP \stackrel{\alpha}{=} \mathbf{new}\ \hat{y} : T'@z'\ \mathbf{in}\ \hat{LP}$, where $\hat{y} = \sigma y$ etc.

- $\Gamma, x : T@z, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta]{c?v} \hat{LQ}$, by (LTS-NEW).
- $x \neq \hat{y}$; moreover, as $x, \hat{y}$ are of extensible types $z, z', x, \hat{y}$ are distinct. This means $\Gamma, \hat{y} : T'@z', x : T@z \equiv \Gamma, x : T@z, \hat{y} : T'@z'$, by Lemma B.1.1.
- $\Gamma, x : T@z, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta]{c?v} \hat{LQ}$, by Lemma B.2.2 (Lts Perm).
- $\Gamma, \hat{y} : T'@z' \Vdash \hat{LP} \xrightarrow[\Delta, x:T@z]{c?v} \hat{LQ}$, by the induction hypothesis.
- $\Gamma \Vdash \mathtt{new}\ y : T'@z'\ \mathtt{in}\ LP \xrightarrow[\Delta, x:T@z]{c?v} \mathtt{new}\ y : T'@z'\ \mathtt{in}\ LQ$, by (LTS-NEW) and alpha-conversion.

Thus true for this case. ■

Lemma B.2.9 (Shifting: input transitions (4.5.6))
Given that Θ is an extensible context with $a, c \notin \mathsf{dom}(\Theta)$ and $\mathsf{dom}(\Theta) \subseteq \mathsf{fv}(v)$, we have:

1. $\Gamma \vdash_a P$ and $\vdash_{\mathsf{L}} \Gamma, \Theta$ implies $\Gamma \Vdash_a\ P \xrightarrow[\Delta, \Theta]{c?v} LP\ \Leftrightarrow\ (\Gamma, \Theta) \Vdash_a\ P \xrightarrow[\Delta]{c?v} LP$; and
2. $\Gamma \vdash LP$ and $\vdash_{\mathsf{L}} \Gamma, \Theta$ implies $\Gamma \Vdash LP \xrightarrow[\Delta, \Theta]{@_a c?v} LQ\ \Leftrightarrow\ (\Gamma, \Theta) \Vdash LP \xrightarrow[\Delta]{@_a c?v} LQ$.

Proof: An induction on the size of Θ, applying Lemma B.2.8. ■

Lemma B.2.10
If $\Gamma \vdash LP$ then $\mathsf{mayMove}(LP) \subseteq \mathsf{mov}(\Gamma)$ and $\mathsf{mayMove}(LP) \subseteq \mathsf{agents}(LP)$.

Lemma B.2.11
If $\Gamma \Vdash LP \xrightarrow[\Delta]{@_a \mathsf{migrate\ to}\ s} LQ$ then $a \in \mathsf{mayMove}(LP)$.

Lemma B.2.12 (Transition preserves mayMove(LP))
Given that Γ is closed located type context, if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then either

- β is an output label and $\mathsf{mayMove}(LQ) \subseteq \mathsf{mayMove}(LP) \cup \mathsf{mov}(\Delta)$; or
- β is not an output label and $\mathsf{mayMove}(LQ) \subseteq \mathsf{mayMove}(LP)$.

Proof: A routine induction on the transition derivation. ■

B.3 Subject Reduction

Theorem B.3.1 (Subject reduction (4.5.1))
Given a closed located type context Γ,

1. if $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ then $\Gamma, \Delta \vdash LP$; and
2. if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\Gamma, \Delta \vdash LQ$.

Proof: First, we prove that if $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ then $\Gamma \vdash_a P$ and $\alpha = \mathsf{migrate\ to}\ s$ implies $\Gamma \vdash s \in \mathtt{Site}$ (required for (Lts-Bound-Migrate) case). This is proved by an induction on derivation of $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$. We shall only demonstrate interesting cases.

Base Case (Lts-L-Let): Supposing

$$\Gamma \Vdash_a \mathbf{let}\ p = ev\ \mathbf{in}\ P \xrightarrow{\tau} @_a \mathsf{match}(p, \mathtt{eval}(ev))P$$

and only (Lts-L-Let) is used for deriving this, we may derive the following. By (Lts-L-Let), $\Gamma \vdash_a \mathbf{let}\ p = ev\ \mathbf{in}\ P$ and $\mathtt{eval}(ev)$ is defined. $\Gamma \vdash ev \in T$, $\Gamma \vdash p \in T \triangleright \Delta$, and $\Gamma, \Delta \vdash_a P$, by (Let). $\Gamma \vdash \mathtt{eval}(ev) \in T$, by Lemma 3.7.3. $\mathsf{match}(p, \mathtt{eval}(ev))$ is defined, by Lemma 3.8.3. Moreover, $\Gamma \vdash_a \mathsf{match}(p, \mathtt{eval}(ev))P$, by Lemma B.1.9. $\Gamma \vdash @_a \mathsf{match}(p, \mathtt{eval}(ev))P$, by (At). Thus true for this case.

Base Case (Lts-L-Create): Supposing

$$\Gamma \Vdash_a \mathbf{create}^Z\ b = P\ \mathbf{in}\ Q \xrightarrow{\tau} \mathbf{new}\ b : \mathtt{Agent}^Z\ \mathbf{in}\ @_a Q \mid @_b P$$

and only (Lts-L-Create) is used for deriving this, we may derive the following. By (Lts-L-Create), $\Gamma \vdash_a \mathbf{create}^Z\ b = P\ \mathbf{in}\ Q$ and $\Gamma \vdash a@s$. $a \neq b$, $\Gamma, b : \mathtt{Agent}^Z \vdash_a P$, and $\Gamma, b : \mathtt{Agent}^Z \vdash_a Q$, by (Create). $\Gamma, b : \mathtt{Agent}^Z \vdash @_a Q$, and $\Gamma, b : \mathtt{Agent}^Z \vdash @_b P$. As $\Gamma \vdash a@s$ and $\vdash \Gamma$, we have $\Gamma \vdash s \in \mathtt{Site}$, by (LC-Var). $\Gamma \vdash \mathbf{new}\ b : \mathtt{Agent}^Z @s\ \mathbf{in}\ @_b P \mid @_a Q$, by (LPar) and (NewAgent). Thus true for this case.

Base Case (Lts-L-Migrate): Supposing $\Gamma \Vdash_a \mathbf{migrate\ to}\ s \rightarrow P \xrightarrow{\mathsf{migrate\ to}\ s} @_a P$, and only (Lts-L-Migrate) is used for deriving this, we may derive the following. $\Gamma \vdash_a \mathbf{migrate\ to}\ s \rightarrow P$, by (Lts-L-Migrate). $\Gamma \vdash s \in \mathtt{Site}$, $\Gamma \vdash a \in \mathtt{Agent}^{\mathtt{m}}$ and $\Gamma \vdash_a P$, by (Migrate). $\Gamma \vdash @_a P$, by (At). In this case $\Gamma \vdash s \in \mathtt{Site}$ and

$$\Gamma \Vdash_a \mathbf{migrate\ to}\ s \rightarrow P \xrightarrow{\mathsf{migrate\ to}\ s} @_a P$$

Thus true for this case.

Base Case (Lts-L-IfLocal-True): Supposing

$$\Gamma \Vdash_a \mathbf{iflocal}\ \langle b \rangle c!v\ \mathbf{then}\ P\ \mathbf{else}\ Q \xrightarrow{\tau} @_a P \mid @_b c!v$$

and only (Lts-L-IfLocal-True) is used for deriving this, we may derive the following. $\Gamma \vdash_a \mathbf{iflocal}\ \langle b \rangle c!v\ \mathbf{then}\ P\ \mathbf{else}\ Q$, since the LTS is typed, and $\Gamma \vdash a@s \wedge b@s$ for some s, by (LTS-L-IFLOCAL-TRUE). $\Gamma \vdash b \in \texttt{Agent}^{\texttt{s}}$, $\Gamma \vdash c \in \texttt{\^{}}^{\texttt{w}}T$, $\Gamma \vdash v \in T$, $\Gamma \vdash_a P$ and $\Gamma \vdash_a Q$, by (IFLOCAL). $\Gamma \vdash @_b c\texttt{!}v$ and $\Gamma \vdash @_a P$, by (AT) and (OUT). $\Gamma \vdash @_a P \mid @_b c\texttt{!}v$, by (LPAR). Thus true for this case.

Base Case (Lts-L-(Rep-)In): Supposing

$$\Gamma \Vdash_a \ \texttt{*}c\texttt{?}v \to P \xrightarrow[\Delta]{c\texttt{?}v} @_a((\mathsf{match}(p,v)P) \mid \texttt{*}c\texttt{?}p \to P)$$

and only (LTS-L-(REP-)IN) is used for deriving this, we may derive the following. By (LTS-L-(REP-)IN), $\Gamma \vdash_a \texttt{*}c\texttt{?}p \to P$ and $\Gamma, \Delta \vdash v \in T$, $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v)$. $\Gamma \vdash c \in \texttt{\^{}}^{\texttt{r}}T$, $\Gamma \vdash p \in T \triangleright \Xi$ and $\Gamma, \Xi \vdash_a P$, by ((REP-)IN). $\mathsf{match}(p,v)$ is defined, by Lemma 3.8.3. $\Gamma, \Delta \vdash_a \mathsf{match}(p,v)\ P$, by Lemma B.1.9. $\Gamma, \Delta \vdash @_a\mathsf{match}(p,v)P \mid \texttt{*}c\texttt{?}p \to P$, by (AT). Thus true for this case.

Base Case (Lts-L-LI-Send): Supposing $\Gamma \Vdash_a \ \langle b@?\rangle c\texttt{!}v \xrightarrow{\tau} @_b c\texttt{!}v$, and only (LTS-L-LI-SEND) is used for deriving this, we may derive the following. $\Gamma \vdash_a \langle b@?\rangle c\texttt{!}v$, since the semantics is typed. $\Gamma \vdash a, b \in \texttt{Agent}^{\texttt{s}}$, $\Gamma \vdash c \in \texttt{\^{}}^{\texttt{w}}T$, and $\Gamma \vdash v \in T$, by (SENDLI) and (AT). $\Gamma \vdash @_b c\texttt{!}v$, by (OUT) and (AT). Thus true for this case.

Therefore subject reduction is true for basic processes. We prove that if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\Gamma \vdash LQ$ and $\beta = @_a\mathsf{migrate\ to}\ s$ implies $\Gamma \vdash s \in \texttt{Site}$. This is proved by an induction on the derivation of $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$.

Base Case (Lts-Local): Supposing $\Gamma \Vdash @_a P \xrightarrow[\Delta]{\beta} LQ$, and (LTS-LOCAL) is used last for deriving this, we may derive the following. $\Gamma \vdash @_a P$, since the semantics is typed, and $\Gamma \Vdash_a \ P \xrightarrow[\Delta]{\alpha} LQ$, by (LTS-LOCAL), where $\beta = \alpha = \tau$ or $\beta = @_a\alpha$. $\Gamma \vdash_a P$, by (AT). $\Gamma, \Delta \vdash LQ$, by Theorem 4.5.1 (Subj) for basic process. If $\alpha = \mathsf{migrate\ to}\ s$ then $\Gamma \vdash s \in \texttt{Site}$, by the proved hypothesis of basic process case. Thus simply true for this case.

Inductive Case (Lts-Cong-R): Supposing $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LR$ and (LTS-CONG-R) is used for deriving the transition, we may derive the following. $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ and $LQ \equiv LR$, for some LQ, by (LTS-CONG-R). $\Gamma, \Delta \vdash LQ$, by the induction hypothesis. $LR \equiv LQ$ implies $\Gamma, \Delta \vdash LR$, by Lemma 4.1.2 (Type StrCong). If $\beta = @_a\mathsf{migrate\ to}\ s$ then $\Gamma \vdash s \in \texttt{Site}$, by the induction hypothesis. Thus true for this case.

Inductive Case (Lts-Comm): Supposing $\Gamma \Vdash LP|LQ \xrightarrow{\tau} \textbf{new}\ \Delta\ \textbf{in}\ LP'|LQ'$ and (LTS-COMM) is used last for deriving the transition, we may derive the following. $\Gamma \Vdash LP \xrightarrow[\Delta]{@_a c\texttt{!}v} LP'$ and $\Gamma \Vdash LQ \xrightarrow[\Delta]{@_a c\texttt{?}v} LQ'$, for some a, c, v, by (LTS-COMM). $\Gamma, \Delta \vdash LP'$ and $\Gamma, \Delta \vdash LQ'$, by the induction hypothesis. $\Gamma, \Delta \vdash LP'|LQ'$, by (LPAR). $\Gamma \vdash \textbf{new}\ \Delta\ \textbf{in}\ LP \mid LQ$, by (NEWAGENT) and (NEWCHANNEL). Thus true for this case.

Inductive Case (Lts-Prl): Supposing $\Gamma \Vdash LP|LQ \xrightarrow[\Delta]{\beta} LR|LQ$ and (Lts-Prl) is used last for deriving the transition, we may derive the following. $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LR$, by (Lts-Prl). $\Gamma, \Delta \vdash LR$, by the induction hypothesis. $\vdash \Gamma, \Delta$, by the definition of the transition relation. $\Gamma, \Delta \vdash LQ$, by Lemma 3.7.2. $\Gamma, \Delta \vdash LR|LQ$, by (LPar). If $\beta = @_a\mathsf{migrate\ to}\ s$ then $\Gamma \vdash s \in \mathtt{Site}$, by the induction hypothesis. Thus true for this case.

Inductive Case (Lts-Open): Supposing $\Gamma \Vdash \mathbf{new}\ x : T@z\ \mathbf{in}\ LP \xrightarrow[\Delta, x:T@z]{@_a c!v} LQ$ and (Lts-Open) is used last for deriving the transition, we may derive the following. By (Lts-Open), we have: $x \in \mathsf{fv}(v)$, $x \neq a, c$, and $\Gamma, x : T@z \Vdash LP \xrightarrow[\Delta]{@_a c!v} LQ$. $\Gamma, x : T@z, \Delta \vdash LQ$, by the induction hypothesis. By 3.7.1, $\Gamma, \Delta, x : T@z \vdash LQ$. Thus true for this case.

Inductive Case (Lts-New): Supposing $\Gamma \Vdash \mathbf{new}\ x : \mathtt{Agent}^Z@s\ \mathbf{in}\ LP \xrightarrow[\Delta]{\beta} LQ$ and (Lts-New) is used last in deriving the transition, we may derive the following. By (Lts-New), $\Gamma, x : \mathtt{Agent}^Z@s \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ with $x \notin \mathsf{fv}(\beta)$. $\Gamma, x : T, \Delta \vdash LQ$, by the induction hypothesis. $\Gamma, \Delta, x : T \vdash LQ$, by 3.7.1. $\Gamma, \Delta \vdash \mathbf{new}\ x : T@s\ \mathbf{in}\ LQ$, by (NewAgent). If $\beta = @_a\mathsf{migrate\ to}\ s$ then $\Gamma \vdash s \in \mathtt{Site}$, by the induction hypothesis. Thus true for this case. The case where x is of channel type is similar.

Inductive Case (Lts-Bound-Mig): Supposing

$$\Gamma \Vdash \mathbf{new}\ a : \mathtt{Agent}^{\mathtt{m}}@s\ \mathbf{in}\ LP \xrightarrow{\tau} \mathbf{new}\ a : \mathtt{Agent}^{\mathtt{m}}@s'\ \mathbf{in}\ LQ$$

and (Lts-Bound-Mig) is used last in deriving the transition, we may derive the following. $\Gamma, a : \mathtt{Agent}^{\mathtt{m}}@s \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s'} LQ$, by (Lts-Bound-Mig). $\Gamma, a : \mathtt{Agent}^Z \vdash LQ$, by the induction hypothesis. $\Gamma \vdash s' \in \mathtt{Site}$, by the claim proved above. $\Gamma \vdash \mathbf{new}\ x : T@s'\ \mathbf{in}\ LQ$, by (NewAgent). Thus true for this case.

Therefore the subject reduction is proved by induction. ■

B.4 (Cong-L) Absorption

Theorem B.4.1 (Cong-L absorption (4.5.2))
Given a closed located type context Γ,

1. if $P \equiv Q$ and $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ then $\Gamma \Vdash_a Q \xrightarrow[\Delta]{\alpha} LP$; and
2. if $LP \equiv LR$ and $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\Gamma \Vdash LR \xrightarrow[\Delta]{\beta} LQ$.

Proof: First of all, we need to make explicit the structural congruence rules in Figure B.1, that ensure that $\equiv$ is indeed a congruence.

$$\text{(Str-Refl)}\quad P \equiv P \qquad \text{(Str-Sym)}\ \frac{P \equiv Q}{Q \equiv P} \qquad \text{(Str-Trans)}\ \frac{P \equiv Q \quad Q \equiv R}{P \equiv R} \qquad \text{(Str-L-Context)}\ \frac{P \equiv Q}{\mathcal{E}_B[P] \equiv \mathcal{E}_B[Q]}$$

$$\text{(Str-L-Refl)}\quad LP \equiv LP \qquad \text{(Str-L-Sym)}\ \frac{LP \equiv LQ}{LQ \equiv LP} \qquad \text{(Str-L-Trans)}\ \frac{LP \equiv LQ \quad LQ \equiv LR}{LP \equiv LR} \qquad \text{(Str-Context)}\ \frac{LP \equiv LQ}{\mathcal{E}[LP] \equiv \mathcal{E}[LQ]}$$

Figure B.1: Supplementary structural congruence rules

The basic process contexts, ranged over by $\mathcal{E}_B[\cdot]$, are defined by the following grammar:

$$\begin{aligned}
\mathcal{E}_B[\cdot] \quad ::= \quad & .\ |\ \mathcal{E}_B[\cdot]|Q\ |\ \mathtt{new}\ c:T\ \mathtt{in}\ \mathcal{E}_B[\cdot]\ |\ c\mathtt{?}p\!\to\!\mathcal{E}_B[\cdot]\ |\ \mathtt{*}c\mathtt{?}p\!\to\!\mathcal{E}_B[\cdot] \\
& |\ \mathtt{if}\ v\ \mathtt{then}\ \mathcal{E}_B[\cdot]\ \mathtt{else}\ Q\ |\ \mathtt{if}\ v\ \mathtt{then}\ P\ \mathtt{else}\ \mathcal{E}_B[\cdot] \\
& |\ \mathtt{create}^{\mathtt{m}}\ a = \mathcal{E}_B[\cdot]\ \mathtt{in}\ Q\ |\ \mathtt{create}^{\mathtt{m}}\ a = P\ \mathtt{in}\ \mathcal{E}_B[\cdot] \\
& |\ \mathtt{migrate\ to}\ s \to \mathcal{E}_B[\cdot]\ |\ \mathtt{let}\ p = ev\ \mathtt{in}\ \mathcal{E}_B[\cdot] \\
& |\ \mathtt{iflocal}\ \langle a\rangle c\mathtt{!}v\ \mathtt{then}\ \mathcal{E}_B[\cdot]\ \mathtt{else}\ Q\ |\ \mathtt{iflocal}\ \langle a\rangle c\mathtt{!}v\ \mathtt{then}\ P\ \mathtt{else}\ \mathcal{E}_B[\cdot]
\end{aligned}$$

Most of this proof is similar to that of [Sew00], we shall only highlight rules which are specific to our setting ie. (Str-Locate) and (Str-Distr).

Base Case (Str-Locate): Supposing $@_a P \equiv @_a Q$ and (Str-Locate) is used last for deriving this equivalence. Let $\Gamma \Vdash @_a P \xrightarrow[\Delta]{\beta} LP$, we may derive the following. $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$, where $\beta = \alpha = \tau$ or $\beta = @_a\alpha$, by (Lts-Local). $\Gamma \Vdash_a Q \xrightarrow[\Delta]{\alpha} LP$, by Theorem 4.5.2 for basic process. $\Gamma \Vdash @_a Q \xrightarrow[\Delta]{\beta} LP$, by (Lts-Local). Thus true for this case.

Base Case (Str-Distr): In this case, $@_a(P|Q) \equiv @_a P|@_a Q$. From left-to-right direction, supposing $\Gamma \Vdash @_a(P|Q) \xrightarrow[\Delta]{\beta} LR$, by (Lts-Local), $\Gamma \Vdash_a P|Q \xrightarrow[\Delta]{\alpha} LR$ and $\alpha = \beta = \tau$ or $\beta = @_a\alpha$. By Lemma B.2.3 (Lts Analysis), either of the following hold.

1. (Lts-L-Prl) (Left) There exists LP such that $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ and $LR \equiv LP|@_a Q$.
2. (Lts-L-Prl) (Right) There exists LQ such that $\Gamma \Vdash_a Q \xrightarrow[\Delta]{\alpha} LQ$ and $LR \equiv @_a P|LQ$.
3. (Lts-L-Comm) (Left) There exist c, v, Ξ, LP, LQ such that $\Gamma \Vdash_a P \xrightarrow[\Xi]{c!v} LP$, $\Gamma \Vdash_a Q \xrightarrow[\Xi]{c?v} LQ$, and $LR \equiv \mathtt{new}\ \Xi\ \mathtt{in}\ LP|LQ$.

4. (LTS-L-COMM) (Right) There exist c, v, Ξ, LP, LQ such that $\Gamma \Vdash_a \; P \xrightarrow[\Xi]{c?v} LP, \Gamma \Vdash_a \; Q \xrightarrow[\Xi]{c!v} LQ$, and $LR \equiv \texttt{new}\ \Xi\ \texttt{in}\ LP|LQ$.

From right-to-left direction, supposing $\Gamma \Vdash @_a P|@_a Q \xrightarrow[\Delta]{\beta} LR$, By Lemma B.2.3 (Lts Analysis), either of the following hold.

1. (LTS-PRL) (Left) There exists LP such that $\Gamma \Vdash @_a P \xrightarrow[\Delta]{\alpha} LP$ and $LR \equiv LP|@_a Q$.
2. (LTS-PRL) (Right) There exists LQ such that $\Gamma \Vdash @_a Q \xrightarrow[\Delta]{\alpha} LQ$ and $LR \equiv @_a P|LQ$.
3. (LTS-L-COMM) (Left) There exist c, v, Ξ, LP, LQ such that $\Gamma \Vdash @_a P \xrightarrow[\Xi]{c!v} LP, \Gamma \Vdash @_a Q \xrightarrow[\Xi]{c?v} LQ$, and $LR \equiv \texttt{new}\ \Xi\ \texttt{in}\ LP|LQ$.
4. (LTS-L-COMM) (Right) There exist c, v, Ξ, LP, LQ such that $\Gamma \Vdash @_a P \xrightarrow[\Xi]{c?v} LP, \Gamma \Vdash @_a Q \xrightarrow[\Xi]{c!v} LQ$, and $LR \equiv \texttt{new}\ \Xi\ \texttt{in}\ LP|LQ$.

The cases correspond exactly (1 to 1 etc.) — one may check in each there are matching transitions. Thus true for this case.

■

B.5 Semantics Matching

Theorem B.5.1 (Labelled and reduction semantics matching (4.5.3))
Given a closed located type context Γ, and a located process LP,

$$\Gamma \Vdash LP \to \Gamma' \Vdash LQ \quad \text{IFF} \quad \begin{cases} \Gamma \Vdash LP \xrightarrow{\tau} LQ & \Gamma' = \Gamma \text{ or} \\ \Gamma \Vdash LP \xrightarrow{@_a \textsf{migrate to } s} LQ & \Gamma' = \Gamma \oplus a \mapsto s \end{cases}$$

Proof: We need to the two directions separately — each of which uses an induction on the derivation of the reduction/transition.

Left-to-right direction Hypothesis: $\Gamma \Vdash LP \to \Gamma' \Vdash LQ$ implies either $\Gamma \Vdash LP \xrightarrow{\tau} LQ$ with $\Gamma' = \Gamma$ or, for some s, $\Gamma \Vdash LP \xrightarrow{@_a \textsf{migrate to } s} LQ$ and $\Gamma' = \Gamma \oplus a \mapsto s$. We shall only demonstrate interesting cases.

Base Case (Red-Create): Supposing only (RED-CREATE) is used for deriving the reduction below.

$$\Gamma \Vdash @_a \texttt{create}^Z\ b = P\ \texttt{in}\ Q \to \Gamma \Vdash \texttt{new}\ b : \texttt{Agent}^Z @s\ \texttt{in}\ (@_a Q|@_b P)$$

We may derive the following. $\Gamma \vdash a@s$, by (RED-CREATE). By (LTS-L-CREATE) and (LTS-LOCAL), $\Gamma \Vdash @_a\texttt{create}^Z\ b = P\ \texttt{in}\ Q \quad \xrightarrow{\tau} \quad \texttt{new}\ b : \texttt{Agent}^Z@s\ \texttt{in}\ @_aQ|@_bP$. Thus true for this case.

Base Case (Red-Migrate): Supposing

$$\Gamma \Vdash @_a\texttt{migrate to}\ s \to P \to \Gamma \oplus a \mapsto s \Vdash @_aP$$

and only (RED-MIGRATE) is used for deriving this reduction. We may derive the following. $\Gamma \Vdash @_a\texttt{migrate to}\ s \to P \xrightarrow{@_a\mathsf{migrate\ to}\ s} @_aP$, by (LTS-L-MIGRATE) and (LTS-LOCAL). It is clear from the definition that $\Gamma \oplus a \mapsto s = \Gamma@_a\mathsf{migrate\ to}\ s$. Thus true for this case.

Base Case (Red-Comm): Supposing $\Gamma \Vdash @_a(c!v|c?p \to P) \to \Gamma \Vdash @_a\mathsf{match}(p,v)P$, and only (RED-COMM) is used for deriving this reduction. We may derive the following.

- $\mathsf{match}(p,v)$ is defined, by (RED-COMM).
- $\Gamma \Vdash_a\ c!v \xrightarrow{c!v} @_a\mathbf{0}$, by (LTS-L-OUT).
- $\Gamma \Vdash_a\ c?p \to P \xrightarrow{c?v} @_a\mathsf{match}(p,v)P$, by (LTS-L-IN).
- $\Gamma \Vdash_a\ c!v|c?p \to P \xrightarrow{\tau} @_a\mathbf{0}|@_a\mathsf{match}(p,v)P$, by (LTS-L-COMM).
- $\Gamma \Vdash_a\ c!v|c?p \to P \xrightarrow{\tau} @_a\mathsf{match}(p,v)P$, by (LTS-L-CONG-R).
- $\Gamma \Vdash @_a(c!v|c?p \to P) \xrightarrow{\tau} @_a\mathsf{match}(p,v)P$, by (LTS-LOCAL).

Thus true for this case.

Inductive Case (Red-Prl): Supposing that $\Gamma \Vdash LP|LQ \to \Gamma' \Vdash LR|LQ$ and (RED-PRL) is the last step in deriving this reduction, we may derive the following. $\Gamma \Vdash LP \to \Gamma' \Vdash LR$, by (RED-PRL). This means that either $\Gamma \Vdash LP|LQ \xrightarrow{\tau} LR|LQ$ and $\Gamma' = \Gamma$; or $\Gamma \Vdash LP|LQ \xrightarrow{@_a\mathsf{migrate\ to}\ s} LR|LQ$ and $\Gamma' = \Gamma \oplus a \mapsto s$, by the induction hypothesis and (LTS-PRL). Thus true for this case.

Inductive Case (Red-New): Supposing that (RED-NEW) is the last step in deriving the reduction below.

$$\Gamma \Vdash \texttt{new}\ x : T@z\ \texttt{in}\ LP \to \Gamma' \Vdash \texttt{new}\ x : T@z'\ \texttt{in}\ LQ$$

We may derive the following. $\Gamma, x : T@z \Vdash LP \to \Xi \Vdash LQ$, by (RED-NEW). The following three cases are all the possibilities of applying the induction hypothesis.

- If $(\Gamma, x : T@z) \Vdash LP \xrightarrow{\tau} LQ$ then, by (LTS-NEW), we have:

$$\Gamma \Vdash \texttt{new}\ x : T@z\ \texttt{in}\ LP \xrightarrow{\tau} \texttt{new}\ x : T@z\ \texttt{in}\ LQ$$

In this case, $\Gamma' = \Gamma$.

- If $(\Gamma, x : T@z) \Vdash LP \xrightarrow{@_a\mathsf{migrate\ to}\ s} LQ$ and $a \neq x$ then, by (Lts-New), we have:

$$\Gamma \Vdash \mathbf{new}\ x : T@z\ \mathbf{in}\ LP \xrightarrow{@_a\mathsf{migrate\ to}\ s} \mathbf{new}\ x : T@z\ \mathbf{in}\ LQ$$

In this case, $\Gamma' = \Gamma \oplus a \mapsto s$.

- If $(\Gamma, x : T@z) \Vdash LP \xrightarrow{@_x\mathsf{migrate\ to}\ s} LQ$ then, by (Lts-bound-Mig), we have:

$$\Gamma \Vdash \mathbf{new}\ x : T@z\ \mathbf{in}\ LP \xrightarrow{\tau} \mathbf{new}\ x : T@s\ \mathbf{in}\ LQ$$

In this case, $\Gamma' = \Gamma$.

Thus true for this case.

Right-to-left direction Hypothesis: $\Gamma \Vdash LP \xrightarrow{\tau} LQ$ implies $\Gamma \Vdash LP \to \Gamma \Vdash LQ$ and $\Gamma \Vdash LP \xrightarrow{@_a\mathsf{migrate\ to}\ s} LQ$ implies $\Gamma \Vdash LP \to \Gamma' \Vdash LQ$, where $\Gamma' = \Gamma \oplus a \mapsto s$. We shall only demonstrate more interesting cases.

Inductive Case (Lts-Comm): Supposing $\Gamma \Vdash LP|LQ \xrightarrow{\tau} \mathbf{new}\ \Xi\ \mathbf{in}\ (LP_1 \mid LQ_1)$, and (Lts-Comm) is the last rule used, we may derive the following.

- $\Gamma \Vdash LP \xrightarrow[\Xi]{@_a c!v} LP_1$ and $\Gamma \Vdash LQ \xrightarrow[\Xi]{@_a c?v} LQ_1$, by (Lts-Comm) and (Lts-Local).
- $LP \equiv \mathbf{new}\ \Delta_1, \Xi\ \mathbf{in}\ (@_a c!v \mid LP')$ and $LQ \equiv \mathbf{new}\ \Delta_2\ \mathbf{in}\ (@_a c?p \to Q \mid LQ')$ (wlog, the case for replicated input is similar) for some $\Delta_1, \Delta_2, LP', p, Q, LQ'$ with $\mathsf{dom}(\Delta_2) \cap \mathsf{dom}(\Xi) = \emptyset$, by Lemma 4.5.4.

  Moreover, $LP_1 \equiv \mathbf{new}\ \Delta_1\ \mathbf{in}\ LP'$ and $LP_2 \equiv \mathbf{new}\ \Delta_2\ \mathbf{in}\ @_a\mathsf{match}(p,v)Q \mid LQ'$.

- Assume (with explicit alpha-conversion if requires) that the domains of Δ_1, Δ_2 are disjoint, we have

$$\mathbf{new}\ \Xi\ \mathbf{in}\ (LP_1 \mid LQ_1) \equiv \mathbf{new}\ \Delta_1, \Delta_2, \Xi\ \mathbf{in}\ (LP' \mid @_a\mathsf{match}(p,v)Q' \mid LQ').$$

- $\mathsf{match}(p,v)$ is defined, by Lemma 3.8.3.
- $\Gamma, \Delta_1, \Delta_2 \Vdash @_a(c!v \mid c?p \to Q) \to \Gamma, \Delta_1, \Delta_2 \Vdash @_a\mathsf{match}(p,v)Q$, by (Red-Comm).
- By (Red-Cong),

$$\Gamma \Vdash \mathbf{new}\ \Delta_1, \Delta_2, \Xi\ \mathbf{in}\ (@_a c!v \mid LP' \mid @_a c?p \to Q \mid LQ')$$
$$\to \Gamma \Vdash \mathbf{new}\ \Delta_1, \Delta_2, \Xi\ \mathbf{in}\ (@_a\mathsf{match}(p,v)Q \mid LP' \mid LQ')$$

- By structural congruence rearrangement, we have:

$$\begin{aligned}&\textbf{new } \Delta_1, \Delta_2, \Xi \textbf{ in } (@_a c!v \mid LP' \mid @_a c?p \to Q \mid LQ')\\ &\quad \equiv (\textbf{new } \Delta_1 \textbf{ in } @_a c!v \mid LP') \mid (\textbf{new } \Delta_2 \textbf{ in } @_a c?p \to Q \mid LQ') \quad \text{and}\\ &\textbf{new } \Delta_1, \Delta_2, \Xi \textbf{ in } (@_a \mathsf{match}(p, v)Q \mid LP' \mid LQ')\\ &\quad \equiv \textbf{new } \Xi \textbf{ in } (\textbf{new } \Delta_1 \textbf{ in } \mid LP') \mid (\textbf{new } \Delta_2 \textbf{ in } @_a \mathsf{match}(p, v)Q \mid LQ')\end{aligned}$$

- That is $\Gamma \Vdash LP \mid LQ \to \Gamma \Vdash \textbf{new } \Xi \textbf{ in } (LP_1 \mid LQ_1)$

Thus true for this case.

Inductive Case (Lts-Prl): Supposing $\Gamma \Vdash LP|LQ \xrightarrow{@_a \mathsf{migrate\ to}\ s} LR|LQ$ and (LTS-PRL) is used as the last rule in derivating the transition, we may derive the following. $\Gamma \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s} LR$, by (LTS-PRL). $\Gamma \Vdash LP \to \Gamma \oplus a \mapsto s \Vdash LR$, by the induction hypothesis. $\Gamma \Vdash LP|LQ \to \Gamma \oplus a \mapsto s \Vdash LR|LQ$, by (RED-PRL). Thus true for this case.

Inductive Case (Lts-Bound-Mig): Supposing

$$\Gamma \Vdash \textbf{new } a : \mathtt{Agent}^{\mathtt{m}}@z \textbf{ in } LP \xrightarrow{\tau} \textbf{new } x : \mathtt{Agent}^{\mathtt{m}}@s \textbf{ in } LR$$

and (LTS-BOUND-MIG) is the last rule used in deriving the transition, we may derive the following. By (LTS-NEW), $\Gamma, a : \mathtt{Agent}^{\mathtt{m}}@z \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s} LR$. By the induction hypothesis, $\Gamma, a : \mathtt{Agent}^{\mathtt{m}}@z \Vdash LP \to \Gamma \oplus a \mapsto \mathtt{Agent}^{\mathtt{s}}@z \Vdash LR$. By (RED-NEW), we have:

$$\Gamma \Vdash \textbf{new } a : \mathtt{Agent}^{\mathtt{s}}@z \textbf{ in } LP \to \Gamma \Vdash \textbf{new } a : \mathtt{Agent}^{\mathtt{s}}@s \textbf{ in } LR$$

Thus true for this case.

The right-to-left direction is thus proved by induction. Hence the lemma. ■

B.6 Translocating Relations

Lemma B.6.1 (Relocation preserves transloc. equivalences)
If $LP \dot{\sim}^M_\Gamma LQ$ then, for any valid relocator δ for (Γ, M), $LP \dot{\sim}^M_{\Gamma\delta} LQ$. Similarly for $\dot{\approx}$ and $\dot{\succeq}$.

Lemma B.6.2 (Strengthening translocating index)
If $LP \dot{\sim}^M_\Gamma LQ$ and $M' \subseteq M$ then $LP \dot{\sim}^{M'}_\Gamma LQ$. Similarly for $\dot{\approx}$ and $\dot{\succeq}$.

Lemma B.6.3
Given that Θ is extensible, $\vdash \Gamma, \Theta$ with $M \subseteq \mathsf{mov}(\Gamma)$ and $M' \subseteq \mathsf{mov}(\Theta)$, if δ is a valid relocator for (Γ, M) then it is also valid for $((\Gamma, \Theta), (M \cup M'))$. Moreover, if δ' is a valid relocator for $((\Gamma, \Theta), (M \cup M'))$ then δ, constructed from δ' by restricting the domain to $\mathsf{dom}(\Gamma)$, is a valid relocator for (Γ, M).

Lemma B.6.4
If $LP \dot{\sim}^M_\Gamma LQ$ and $\Gamma \equiv \Theta$ then $LP \dot{\sim}^M_\Theta LQ$. Similarly for $\dot{\approx}$ and $\dot{\succeq}$.

Lemma B.6.5 (Injection preserves translocating equivalences (5.3.2))
Given that Γ is a closed located type context and $\rho : \mathsf{dom}(\Gamma) \to \mathcal{X}$ is injective, we have:

1. if $LP_1 \dot{\sim}^M_\Gamma LP_2$ then $\rho LP_1 \dot{\sim}^{\rho M}_{\rho\Gamma} \rho LP_2$; and
2. if $LP_1 \dot{\approx}^M_\Gamma LP_2$ then $\rho LP_1 \dot{\approx}^{\rho M}_{\rho\Gamma} \rho LP_2$.

Proof: Assuming $\mathcal{S}$ is a translocating strong bisimulation, construct a translocating relation $\mathcal{R}s$ with $\mathcal{R}s^{M'}_\Theta$ is defined as follows:

$$\left\{(LP', LQ') \;\middle|\; \exists \begin{array}{l} LP, LQ, \Phi, M'', \\ \text{injective } \sigma : \mathsf{dom}(\Phi) \to \mathcal{X} \end{array} . \begin{array}{l} LP' = \sigma LP \wedge LQ' = \sigma LQ \wedge \\ LP \mathcal{S}^{M''}_\Phi LQ \wedge \Theta = \sigma\Phi \wedge M' = \sigma M'' \end{array} \right\}$$

We shall check that this is indeed a strong translocating bisimulation.

Supposing $(LP', LQ') \in \mathcal{R}s^{M'}_\Theta$, by definition of $\mathcal{R}s$, there exist LP, LQ, Φ, M'' and an injective substitution σ such that $LP' = \sigma LP$, $LQ' = \sigma LQ$, $LP\mathcal{S}^{M''}_\Phi LQ$, $\Theta = \sigma\Phi$ and $M' = \sigma M''$. Let δ be a valid relocator for (Φ, M'') (this implies $\sigma\delta$ is a valid relocator for (Θ, M')) and $\Theta(\sigma\delta) \Vdash LP' \xrightarrow[\Delta]{\beta} LP_1$,

- In order to make sure the names in Δ do not clash with that in Φ, we pick an injective substitution $\sigma' : \mathsf{dom}(\Delta) \to \mathcal{X}/\mathsf{dom}(\Phi)$. Clearly $\sigma^{-1} + \sigma'$ is a bijection from $\mathsf{dom}(\sigma\Phi, \Delta)$ to $\mathsf{dom}(\Phi, \sigma'\Delta)$.
- $\sigma^{-1}(\Theta(\sigma\delta)) \Vdash \sigma^{-1}LP' \xrightarrow[\sigma'\Delta]{(\sigma^{-1}+\sigma')\beta} (\sigma^{-1} + \sigma')LP_1$, by Lemma 4.5.7 (Lts Inj).
  Thus $\Phi\delta \Vdash LP \xrightarrow[\sigma\Delta]{(\sigma^{-1}+\sigma)\beta} (\sigma^{-1} + \sigma)LP'$.
- Since $LP\mathcal{S}^{M''}_\Phi LQ$ and δ is a valid relocator for (Φ, M''), there exists LQ_1 such that $\Phi\delta \Vdash LQ \xrightarrow[\sigma'\Delta]{(\sigma^{-1}+\sigma')\beta} LQ_1$ and $(\sigma^{-1} + \sigma')LP'\mathcal{S}^{M'' \uplus_{(\sigma^{-1}+\sigma')\beta} \mathsf{mov}(\sigma'\Delta)}_{\Phi\delta((\sigma^{-1}+\sigma')\beta), \sigma'\Delta} LQ_1 \quad (\dagger)$.
- Applying $\sigma + (\sigma')^{-1}$, we have $\Theta(\sigma\delta) \Vdash \sigma LQ \xrightarrow[\Delta]{\beta} (\sigma + (\sigma')^{-1})LQ_1 \quad (\dagger)$, by Lemma 4.5.7 (Lts Inj).
- As $(\sigma^{-1} + \sigma')LP'\mathcal{S}^{M'' \uplus_{(\sigma^{-1}+\sigma')\beta} \mathsf{mov}(\sigma'\Delta)}_{\Phi((\sigma^{-1}+\sigma')\beta), \sigma'\Delta} LQ_1$, by definition, we have:

$$LP' \; \mathcal{R}s^{(\sigma+(\sigma')^{-1})(M' \uplus_{(\sigma^{-1}+\sigma')\beta} \mathsf{mov}(\sigma'\Delta))}_{(\sigma+(\sigma')^{-1})(\Phi((\sigma^{-1}+\sigma')\beta), \sigma'\Delta)} \; (\sigma + (\sigma')^{-1})LQ_1$$

- Rearranging, we have: $LP' \; \mathcal{R}s^{(M' \uplus_\beta \mathsf{mov}(\Delta))}_{\Theta\beta, \Delta} \; (\sigma + (\sigma')^{-1})LQ_1$.

Therefore $\mathcal{R}s$ is a translocating strong bisimulation.

Supposing $\mathcal{S}$ translocating weak simulation (expansion etc.) and replacing $\xrightarrow[\Delta]{\beta}$ with $\xRightarrow[\Delta]{\hat{\beta}}$ at (†), we obtain the prove for the weak case. ■

Lemma B.6.6 (Strengthening and weakening: transloc. bisimulations (5.3.5)) Let Γ, Θ be closed located type contexts with Θ extensible and $\vdash_{\mathsf{L}} \Gamma, \Theta$. Let $M_1 \subseteq \mathsf{mov}(\Gamma)$ and $M_2 \subseteq \mathsf{mov}(\Theta)$, then if $\Gamma \vdash LP, LQ$ the following hold.

$$\begin{array}{rcl} LP \dot{\sim}^{M_1 \cup M_2}_{\Gamma,\Theta} LQ & \iff & LP \dot{\sim}^{M_1}_{\Gamma} LQ \\ LP \dot{\approx}^{M_1 \cup M_2}_{\Gamma,\Theta} LQ & \iff & LP \dot{\approx}^{M_1}_{\Gamma} LQ \end{array}$$

Proof: We prove this in two parts, omitting symmetric cases.

Weakening Construct a translocating relation $\mathcal{R}$ with $\mathcal{R}^M_\Psi$ defined as follows:

$$\{(LP_1, LQ_2) \mid \exists \Phi, \Xi, M' \,.\, \Psi = (\Phi, \Xi) \wedge LP_1 \dot{\sim}^{M'}_{\Phi} LQ_2 \wedge M \cap \mathsf{mov}(\Phi) \subseteq M'\}$$

We shall check that this is indeed a strong translocating bisimulation.

Supposing $(LP_1, LQ_2) \in \mathcal{R}^M_\Psi$, this implies there exist Φ, Ξ, M' such that $\Psi = (\Phi, \Xi)$, $LP_1 \dot{\sim}^{M'}_{\Phi} LQ_2$ and $M \cap \mathsf{mov}(\Phi) \subseteq M'$. Let δ be a valid relocator for (Ψ, M) and $\Psi\delta \Vdash LP_1 \xrightarrow[\Delta]{\beta} LP_2$. We only demonstrate the case where $\beta = @_a c\boldsymbol{?}v$, for some a, c, v. The case where β is not an input label is similar, except "shifting" of extruded context never occurs.

- By Lemma B.2.1 (Lts FV), $\{a, c\} \subseteq \mathsf{fv}(LP_1)$. Moreover, as $\Phi \vdash LP_1$, we have $\mathsf{fv}(LP_1) \subseteq \mathsf{dom}(\Phi)$ (the transloc relation is typed) and thus $\{a, c\} \subseteq \mathsf{dom}(\Phi)$, by Lemma B.1.3 (Type FV).
- Let $\Xi \equiv \Xi_1, \Xi_2$ with $\mathsf{dom}(\Xi) \cap \mathsf{fv}(v) = \mathsf{dom}(\Xi_1)$, we have, by Lemma 4.5.5 (Lts SW), $\Phi, \Xi_2\delta \Vdash LP_1 \xrightarrow[\Delta]{@_a c\boldsymbol{?}v} LP_2$.
- By Lemma 4.5.6 (Lts Shift), we have $\Phi\delta \Vdash LP_1 \xrightarrow[\Delta, \Xi_2\delta]{@_a c\boldsymbol{?}v} LP_2$.
- Since $LP_1 \dot{\sim}^{M'}_{\Phi} LQ_2$ and δ restricted to $\mathsf{dom}(\Phi)$ is a valid relocator for (Φ, M'), there exists LQ_2 with $\Phi\delta \Vdash LQ_2 \xrightarrow[\Delta, \Xi_2\delta]{@_a c\boldsymbol{?}v} LQ_2$ (†) and $LP_2 \dot{\sim}^{M' \cup \mathsf{mov}(\Delta, \Xi_2)}_{\Phi, \Delta, \Xi_2\delta} LQ_2$.
- By Lemma 4.5.6 (Lts Shift) and Lemma 4.5.5 (Lts SW), we have $\Xi\delta \Vdash LQ_2 \xrightarrow[\Delta]{@_a c\boldsymbol{?}v} LQ_2$ (†).
- Since $M \cap \mathsf{mov}(\Phi) \subseteq M'$, we have $(M \cup \mathsf{mov}(\Delta)) \cap \mathsf{mov}(\Phi) \subseteq M' \cup \mathsf{mov}(\Delta)$.

- This implies $LP_2\ \mathcal{R}^{M\cup\mathsf{mov}(\Delta)}_{\Psi,\Delta}\ LQ_2$.

Thus, for this case, $\mathcal{R}$ is a translocating strong bisimulation.

The proof for the weak translocating simulation can be obtained by replacing $\dot{\sim}$ with $\dot{\approx}$ everywhere and $\xrightarrow[\Xi]{\beta}$ with $\xRightarrow[\Xi]{\hat{\beta}}$ at (†).

Strengthening Construct a translocating relation $\mathcal{S}$ such that $\mathcal{S}^M_\Phi$ is as follows.

$$\{(LP_1, LQ_2) \mid \Phi \vdash LP_1, LQ_2\ \wedge\ \exists \Xi, M'\ .\ LP_1 \dot{\sim}^{M\cup M'}_{\Phi,\Xi} LQ_2\ \wedge\ M \subseteq \mathsf{mov}(\Phi) \wedge\ M' \subseteq \mathsf{mov}(\Xi)\}$$

We shall prove that this is a translocating strong bisimulation.

Supposing $(LP_1, LQ_2) \in \mathcal{S}^M_\Phi$, this implies there exists Ξ, M' such that $LP_1 \dot{\sim}^{M\cup M'}_{\Phi,\Xi} LQ_2$, $M \subseteq \mathsf{mov}(\Phi)$ and $M' \subseteq \mathsf{mov}(\Xi)$. Let δ is a valid relocator for (Φ, M) and $\Phi\delta \Vdash LP_1 \xrightarrow[\Delta]{\beta} LP_2$, we may derive the following.

- To make sure that the names in Δ and those in Ξ do not clash, we pick an injective valid substitution $\rho : \mathsf{dom}(\Xi) \to \mathcal{X}/\mathsf{dom}(\Phi, \Delta)$.
- $LP_1 \dot{\sim}^{M\cup M'}_{\Phi,\Xi} LQ_2$ implies $\vdash \Phi, \Xi$. Furthermore, this means that $\vdash \Phi, \rho\Xi$, $\mathsf{dom}(\rho\Xi) \cap (\mathsf{dom}(\Delta) \cup \mathsf{fv}(\beta)) = \emptyset$ (since $\mathsf{fv}(\beta) \subseteq \mathsf{dom}(\Phi, \Delta)$). By Lemma 4.5.5 (Lts SW), we have: $\Phi\delta, \rho\Xi \Vdash LP_1 \xrightarrow[\Delta]{\beta} LP_2$.
- Since $\Phi \vdash LP_1, LQ_2$, $LP_1 \dot{\sim}^{M\cup\rho M'}_{\Phi,\rho\Xi} LQ_2$, by Lemma 5.3.2 (Transloc Bisim Inj).
- Since δ is a valid relocator for $(\Phi, \rho\Xi, M \cup \rho M')$ (by Lemma B.6.3), there exists LQ_2 such that $\Phi\delta, \rho\Xi \Vdash LQ_2 \xrightarrow[\Delta]{\beta} LQ_2$ and $LP_2\rho \dot{\sim}^{M\cup\rho M'\uplus_\beta \mathsf{mov}(\Delta)}_{\Phi,\rho\Xi,\Delta} LQ_2$ (†), by the definition of translocating equivalence.
- $\Phi, \rho\Xi, \Delta \equiv \Phi, \Delta, \rho\Xi$ since $\Delta, \rho\Xi$ are extensible.
- By the definition of $\mathcal{S}$, $LP_2\ \mathcal{S}^{M\uplus_\beta \mathsf{mov}(\Delta)}_{\Phi,\Delta}\ \rho^{-1}LQ_2$ (pick $\rho\Xi, \rho M'$).

Therefore $\mathcal{S}$ is a translocating strong bisimulation.

The proof for the weak translocating simulation can be obtained by replacing $\dot{\sim}$ with $\dot{\approx}$ everywhere and $\xrightarrow[\Xi]{\beta}$ with $\xRightarrow[\Xi]{\hat{\beta}}$ at (†).

Therefore the lemma is verified. ■

B.7 Congruence Result

Theorem B.7.1 (Composing translocating bisimulations (5.3.6))
Let Γ, Θ be closed located type contexts, with Θ extensible; moreover, let $M_P, M_Q \subseteq \mathsf{mov}(\Gamma, \Theta)$, and suppose $\mathsf{mayMove}(LQ, LQ') \subseteq M_P$ and $\mathsf{mayMove}(LP, LP') \subseteq M_Q$.

- $LP \dot{\sim}^{M_P}_{\Gamma,\Theta} LP'$ and $LQ \dot{\sim}^{M_Q}_{\Gamma,\Theta} LQ'$ implies:

$$\texttt{new}\ \Theta\ \texttt{in}\ (LP \mid LQ)\ \dot{\sim}^{M_P \cap M_Q \cap \mathsf{mov}(\Gamma)}_{\Gamma}\ \texttt{new}\ \Theta\ \texttt{in}\ \big(LP' \mid LQ'\big)\ \text{; and}$$

- $LP \dot{\approx}^{M_P}_{\Gamma,\Theta} LP'$ and $LQ \dot{\approx}^{M_Q}_{\Gamma,\Theta} LQ'$ implies:

$$\texttt{new}\ \Theta\ \texttt{in}\ (LP \mid LQ)\ \dot{\approx}^{M_P \cap M_Q \cap \mathsf{mov}(\Gamma)}_{\Gamma}\ \texttt{new}\ \Theta\ \texttt{in}\ \big(LP' \mid LQ'\big)\,.$$

Proof: We construct a translocating relation $\mathcal{R}$ with $\mathcal{R}^M_\Psi$ defined

$$\{(\texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_k \mid LQ_k),\ \ \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_k \mid LQ'_k)) \mid \text{side-condition}\}$$

where the side-condition is that there exists closed located type context Γ and extensible located type contexts $\Delta_{ex}, \Theta_{in}, \Theta^{LP}_{out}, \Theta^{LQ}_{out}, \Theta^{LP}_{comm}, \Theta^{LQ}_{comm}$ and sets of names M_P, M_Q such that:

$$\begin{aligned}
&\Psi \equiv \Gamma, \Delta_{ex}, \Theta_{in}, \Theta_{out} && ([\mathrm{TC}])\\
&M = (\mathsf{mov}(\Gamma) \cap M_P \cap M_Q) \cup \mathsf{mov}(\Theta_{in}) && ([\mathrm{TL}])\\
&\mathsf{mayMove}(LQ_k, LQ'_k) \subseteq M_P \cup \mathsf{mov}(\Theta^{LQ}) && ([\mathrm{Mov}LP])\\
&\mathsf{mayMove}(LP_k, LP'_k) \subseteq M_Q \cup \mathsf{mov}(\Theta^{LP}) && ([\mathrm{Mov}LQ])\\
&LP_k \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in}, \Theta^{LQ})}_{\Psi, \Delta, \Theta_{comm}} LP'_k && ([\mathrm{Bisim}LP])\\
&LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Theta_{in}, \Theta^{LP})}_{\Psi, \Delta, \Theta_{comm}} LQ'_k && ([\mathrm{Bisim}LQ])
\end{aligned}$$

Here Θ_{comm} denotes $(\Theta^{LP}_{comm}, \Theta^{LQ}_{comm})$, and similarly for Θ_{out}. We also let Θ^{LP} denote $(\Theta^{LP}_{out}, \Theta^{LP}_{comm})$, and similarly for Θ^{LQ}. We shall prove that $\mathcal{R}$ is a translocating strong bisimulation.

Supposing

$$(\texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_k \mid LQ_k),\ \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_k \mid LQ'_k)) \quad \in \quad \mathcal{R}^M_\Psi$$

This means that there exist closed extensible located type contexts $\Gamma, \Delta_{ex}, \Theta_{in}, \Theta^{LP}_{out}, \Theta^{LQ}_{out}, \Theta^{LP}_{comm}, \Theta^{LQ}_{comm}$ and two sets of names M_P, M_Q satisfying the above conditions. Let δ be a valid relocator for (Ψ, M) with $\Psi\delta \Vdash \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_k \mid LQ_k) \xrightarrow[\Delta]{\beta} LR$, then the following are all the possibilities (by Lemma B.2.3 (Lts Analysis)):

1. LP_k or LQ_k commits an action (silent, input, output, migration), and
2. LP_k and LQ_k communicate (LP_k sends and LQ_k receives, or vice versa).

We select the following for demonstration (ignoring the silent transition and symmetric cases). Note that since $\Theta_{comm}, \Theta^{LP}$ etc. are extensible and that if Θ_1, Θ_2 are extensible then $\Theta_1, \Theta_2 \equiv \Theta_2, \Theta_1$ (provided that their domain are distinct and they are well-formed etc.), we can mostly ignore the order of these extensible contexts.

Case $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP_k \xrightarrow[\Xi]{@_a x!v} LP_{k+1}$**:** Let $\Delta \equiv \Delta_1, \Delta_2$ and $\Theta^{L?}_{comm} \equiv \Theta^{L?}_{comm1}, \Theta^{L?}_{comm2}$, where $? = P$ or Q, such that $\mathsf{dom}(\Delta, \Theta_{comm}) \cap \mathsf{fv}(v) = \mathsf{dom}(\Delta_1, \Theta_{comm1})$. In this case, we may derive the following.

- $\Psi\delta \Vdash \mathtt{new}\ \Delta, \Theta_{comm}\ \mathtt{in}\ (LP_k | LQ_k) \xrightarrow[\Xi, \Delta_1, \Theta_{comm1}]{@_a x!v} \mathtt{new}\ \Delta_2, \Theta_{comm2}\ \mathtt{in}\ (LP_{k+1} | LQ_k)$, by (Lts-Open), (Lts-New) and (Lts-Prl).
- By [BisimLP], there exists LP'_{k+1} such that, $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP'_k \xrightarrow[\Xi]{@_a x!v} LP'_{k+1}$ (†) and $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in}, \Theta^{LQ})}_{\Psi\delta, \Delta, \Theta_{comm}, \Xi} LP'_{k+1}$, by the definition of translocating strong bisim.
- $\Psi\delta \Vdash \mathtt{new}\ \Delta, \Theta_{comm}\ \mathtt{in}\ (LP'_k | LQ'_k) \xrightarrow[\Xi, \Delta_1, \Theta_{comm1}]{@_a x!v} \mathtt{new}\ \Delta_2, \Theta_{comm2}\ \mathtt{in}\ (LP'_{k+1} | LQ'_k)$ (†), by (Lts-Open), (Lts-New) and (Lts-Prl).
- Let $\Psi' = \Psi\delta, \Xi, \Delta_1, \Theta_{comm1}$, we have:

  $$\mathtt{new}\ \Delta_2, \Theta_{comm2}\ \mathtt{in}\ (LP_{k+1} | LQ_k)\ \mathcal{R}^M_{\Psi'}\ \mathtt{new}\ \Delta_2, \Theta_{comm2}\ \mathtt{in}\ (LP'_{k+1} | LQ_k)$$

  as there exist

  $$\begin{array}{lcllcl} \Delta'_{ex} & \equiv & \Delta_{ex}\delta, \Delta_1 & \Xi_{in} & \equiv & \Theta_{in}\delta \\ \Xi^{LP}_{out} & \equiv & \Theta^{LP}_{out}, \Theta^{LP}_{comm1}, \Xi & \Xi^{LQ}_{out} & \equiv & \Theta^{LQ}_{out}, \Theta^{LQ}_{comm1} \\ \Xi^{LP}_{comm} & \equiv & \Theta^{LP}_{comm2} & \Xi^{LQ}_{comm} & \equiv & \Theta^{LQ}_{comm2} \end{array}$$

  such that

  1. [TC] $\Psi' \equiv (\Gamma, \Delta_{ex}, \Theta_{in}, \Theta_{out})\delta, \Xi, \Theta_{comm1}, \Delta_1 \equiv \Gamma\delta, \Delta'_{ex}, \Xi_{in}, \Xi_{out}$.
  2. [TL] $M = (\mathsf{mov}(\Gamma) \cap M_P \cap M_Q) \cup \mathsf{mov}(\Xi_{in})$, as $\mathsf{mov}(\Xi_{in}) = \mathsf{mov}(\Theta_{in})$.
  3. [MovLP] $\mathsf{mayMove}(LQ_k, LQ_k) \subseteq M_P \cup \mathsf{mov}(\Xi^{LQ})$ since $\mathsf{dom}(\Theta^{LQ}) = \mathsf{dom}(\Xi^{LQ})$.
  4. [MovLQ] $\mathsf{mayMove}(LP_{k+1}, LP'_{k+1}) \subseteq M_Q \cup \mathsf{mov}(\Xi^{LP})$, by Lemma B.2.12.
  5. [BisimLP] $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in}, \Theta^{LQ})}_{\Psi\delta, \Delta, \Theta_{comm}, \Xi} LP'_{k+1}$ implies, by context rearranging,

     $$LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Xi_{in}, \Xi^{LQ})}_{\Psi', \Delta_2, \Xi_{comm2}} LP'_{k+1}$$

6. [BisimLQ] $LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Theta_{in},\Theta^{LP})}_{\Psi,\Delta,\Theta_{comm}} LQ'_k$ implies,
   - By Lemma 5.3.5 (Transloc Bisim SW), $LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Theta_{in},\Theta^{LP},\Xi)}_{\Psi,\Delta,\Theta_{comm},\Xi} LQ'_k$.
   - Since δ is valid for (Ψ, M), by Lemma B.6.1, and context rearranging, we have: $LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Xi_{in},\Xi^{LP})}_{\Psi',\Delta_2,\Xi_{comm2}} LQ'_k$.

Thus RHS translocating strong simulates LHS in this case.

Case $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP_k \xrightarrow[\Xi]{@_a x?v} LP_{k+1}$ **with** $\mathsf{fv}(v) \cap \mathsf{dom}(\Delta, \Theta_{comm}) = \emptyset$**::** This implies:

- $\Psi\delta \Vdash \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_k | LQ_k) \xrightarrow[\Xi]{@_a x?v} \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_{k+1} | LQ_k)$, by (Lts-New) and (Lts-Prl).
- By [BisimLP], there exists LP'_{k+1} such that, $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP'_k \xrightarrow[\Xi]{@_a x?v} LP'_{k+1}$ (†), and $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in},\Theta^{LQ}) \cup \mathsf{mov}(\Xi)}_{\Psi\delta,\Delta,\Theta_{comm},\Xi} LP'_{k+1}$ by the definition of translocating strong bisim.
- $\Psi\delta \Vdash \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_k | LQ'_k) \xrightarrow[\Xi]{@_a x?v} \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_{k+1} | LQ'_k)$ (†), by (Lts-New) and (Lts-Prl).
- Let $\Psi' \equiv \Psi\delta, \Xi$, we have

  $$\texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_{k+1} | LQ_k)\ \mathcal{R}^{M \cup \mathsf{mov}(\Xi)}_{\Psi'}\ \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_{k+1} | LQ_k)$$

  as there exists $\Theta^{LP}_{out}, \Theta^{LQ}_{out}, \Theta^{LP}_{comm}, \Theta^{LQ}_{comm}$ and

  $$\Delta'_{ex} \equiv \Delta_{ex}\delta \qquad \Xi_{in} \equiv \Theta_{in}\delta, \Xi$$

  such that

  1. [TC] $\Psi' \equiv (\Gamma, \Delta_{ex}, \Theta_{in}\Theta_{out})\delta, \Xi \equiv \Gamma\delta, \Delta'_{ex}, \Xi_{in}, \Theta_{out}$.
  2. [TL] $M \cup \mathsf{mov}(\Xi) = (\mathsf{mov}(\Gamma) \cap M_P \cap M_Q) \cup \mathsf{mov}(\Xi_{in})$, as $\mathsf{mov}(\Xi_{in}) = \mathsf{mov}(\Theta_{in}, \Xi)$.
  3. [MovLP] $\mathsf{mayMove}(LQ_k, LQ'_k) \subseteq M_P \cup \mathsf{mov}(\Theta^{LQ})$ remains valid.
  4. [MovLQ] $\mathsf{mayMove}(LP_{k+1}, LP'_{k+1}) \subseteq M_Q \cup \mathsf{mov}(\Theta^{LP})$, by Lemma B.2.12.
  5. [BisimLP] $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in},\Xi,\Theta^{LQ})}_{\Psi\delta,\Delta,\Theta_{comm},\Xi} LP'_{k+1}$ implies, by rearranging,

     $$LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Xi_{in},\Xi^{LQ})}_{\Psi',\Delta,\Theta_{comm}} LP'_{k+1}$$

  6. [BisimLQ] Since δ is valid for (Ψ, M), $LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Theta_{in},\Theta^{LP})}_{\Psi,\Delta,\Theta_{comm}} LQ'_k$ implies, by Lemma 5.3.5 (Transloc Bisim SW) and Lemma B.6.1,

     $$LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Theta_{in},\Theta^{LP},\Xi)}_{\Psi',\Delta,\Theta_{comm}} LQ'_k.$$

Thus $LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Xi_{in}, \Theta^{LP})}_{\Psi', \Delta, \Theta_{comm}} LQ'_k$.

Thus RHS translocating strong simulates LHS in this case.

Case $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP_k \xrightarrow[\Xi]{@_a x\mathbf{!}v} LP_{k+1}$ **and** $\Psi\delta, \Delta, \Theta_{comm} \Vdash LQ_k \xrightarrow[\Xi]{@_a x\mathbf{?}v} LQ_{k+1}$**:** assuming Ξ is fresh. This implies:

- $\Psi\delta \Vdash$ **new** Δ, Θ_{comm} **in** $(LP_k | LQ_k) \xrightarrow{\tau}$ **new** $\Delta, \Theta_{comm}, \Xi$ **in** $(LP_{k+1} | LQ_{k+1})$, by (Lts-New) and (Lts-Comm).
- By [BisimLP], there exists LP'_{k+1} such that, $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP' \xrightarrow[\Xi]{@_a x\mathbf{!}v} LP'_{k+1}$ (†), and $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in}, \Theta^{LQ})}_{\Psi\delta, \Delta, \Theta_{comm}, \Xi} LP'_{k+1}$ by the definition of translocating strong bisim.
- By [BisimLP], there exists LQ'_{k+1} such that, $\Psi\delta, \Delta, \Theta_{comm} \Vdash LQ' \xrightarrow[\Xi]{@_a x\mathbf{?}v} LQ'_{k+1}$ (†), and $LQ_{k+1} \dot{\sim}^{M_Q \cup \mathsf{mov}(\Theta_{in}, \Theta^{LP}) \cup \mathsf{mov}(\Xi)}_{\Psi\delta, \Delta, \Theta_{comm}, \Xi} LQ'_{k+1}$ by the definition of translocating strong bisim.
- $\Psi\delta \Vdash$ **new** Δ, Θ_{comm} **in** $(LP'_k | LQ'_k) \xrightarrow{\tau}$ **new** $\Delta, \Theta_{comm}, \Xi$ **in** $(LP'_{k+1} | LQ'_{k+1})$ (†), by (Lts-New) and (Lts-Comm).
- Let $\Psi' = \Psi\delta$, we have:

  $$\textbf{new } \Delta, \Theta_{comm}, \Xi \textbf{ in } (LP_{k+1} | LQ_{k+1}) \; \mathcal{R}^M_{\Psi'} \; \textbf{new } \Delta, \Theta_{comm}, \Xi \textbf{ in } (LP'_{k+1} | LQ'_{k+1})$$

  as there exists $\Theta^{LP}_{out}, \Theta^{LQ}_{out}$ and

  $$\begin{array}{lclllcl} \Delta'_{ex} & \equiv & \Delta_{ex}\delta & & \Xi_{in} & \equiv & \Theta_{in}\delta \\ \Xi^{LP}_{comm} & \equiv & \Theta^{LP}_{comm}, \Xi & & \Xi^{LQ}_{comm} & \equiv & \Theta^{LQ}_{comm} \end{array}$$

  such that

  1. [TC] $\Psi' \equiv (\Gamma, \Delta_{ex}, \Theta_{in}\Theta^{LP}_{out}, \Theta^{LQ}_{out})\delta \equiv \Gamma, \Delta'_{ex}, \Xi_{in}, \Theta_{out}$.
  2. [TL] $M = (\mathsf{mov}(\Gamma) \cap M_P \cap M_Q) \cup \mathsf{mov}(\Xi_{in})$, as $\mathsf{mov}(\Xi_{in}) = \mathsf{mov}(\Theta_{in})$.
  3. [MovLP] $\mathsf{mayMove}(LQ_{k+1}, LQ'_{k+1}) \subseteq M_P \cup \mathsf{mov}(\Xi^{LQ})$, by Lemma B.2.12.
  4. [MovLQ] $\mathsf{mayMove}(LP_{k+1}, LP'_{k+1}) \subseteq M_Q \cup \mathsf{mov}(\Xi^{LP})$, by Lemma B.2.12.
  5. [BisimLP] $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in}, \Theta, \Theta^{LQ})}_{\Psi\delta, \Delta, \Theta_{comm}, \Xi} LP'_{k+1}$ implies, by rearranging,

  $$LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Xi_{in}, \Xi^{LQ})}_{\Psi', \Delta, \Xi_{comm}} LP'_{k+1}$$

6. [BisimLQ] $LQ_{k+1} \dot{\sim}^{M_Q \cup \mathsf{mov}(\Theta_{in}, \Theta^{LP}, \Xi)}_{\Psi\delta, \Delta, \Theta_{comm}, \Xi} LQ'_{k+1}$ implies, by rearranging,

$$LQ_{k+1} \dot{\sim}^{M_Q \cup \mathsf{mov}(\Xi_{in}, \Xi^{LP})}_{\Psi', \Delta, \Xi_{comm}} LQ'_{k+1}$$

Thus RHS translocating strong simulates LHS in this case.

Case $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP_k \xrightarrow{@_a \textbf{migrate to } s} LP_{k+1}$ **and** $a \in \mathsf{dom}(\Gamma)$**:** This implies:

- $\Psi\delta \Vdash \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_k | LQ_k) \xrightarrow{@_a \mathsf{migrate\ to}\ s} \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_{k+1} | LQ_k)$, by (Lts-New) and (Lts-Prl).
- By [BisimLP], there exists LP'_{k+1} such that, $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP'_k \xrightarrow{@_a \mathsf{migrate\ to}\ s} LP'_{k+1}$ (†), and $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in}, \Theta^{LQ})}_{\Psi\delta @_a \mathsf{migrate\ to}\ s, \Delta, \Theta_{comm}} LP'_{k+1}$ by the definition of translocating strong bisim.
- $\Psi\delta \Vdash \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_k | LQ'_k) \xrightarrow{@_a \mathsf{migrate\ to}\ s} \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_{k+1} | LQ'_k)$ (†), by (Lts-New) and (Lts-Prl).
- Let $\Psi' = \Psi\delta @_a \mathsf{migrate\ to}\ s$, we have:

$$\texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_{k+1} | LQ_k)\ \mathcal{R}^M_{\Psi'}\ \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_{k+1} | LQ_k)$$

  as there exists

$$\Delta'_{ex} \equiv \Delta_{ex}\delta \qquad \Xi_{in} \equiv \Theta_{in}\delta$$

  and $\Theta^{LP}_{out}, \Theta^{LQ}_{out}, \Theta^{LP}_{comm}, \Theta^{LQ}_{comm}$ such that

  1. [TC] $\Psi' \equiv (\Gamma, \Delta_{ex}, \Theta_{in}, \Theta_{out})\delta @_a \mathsf{migrate\ to}\ s \equiv \Gamma\delta @_a \mathsf{migrate\ to}\ s, \Delta'_{ex}, \Xi_{in}, \Theta_{out}$.
  2. [TL] $M = (\mathsf{mov}(\Gamma) \cap M_P \cap M_Q) \cup \mathsf{mov}(\Xi_{in})$, as $\mathsf{mov}(\Xi_{in}) = \mathsf{mov}(\Theta_{in})$.
  3. [MovLP] $\mathsf{mayMove}(LQ_k, LQ'_k) \subseteq M_P \cup \mathsf{mov}(\Theta^{LQ})$ remains valid.
  4. [MovLQ] $\mathsf{mayMove}(LP_{k+1}, LP'_{k+1}) \subseteq M_Q \cup \mathsf{mov}(\Theta^{LP})$, by Lemma B.2.12.
  5. [BisimLP] $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Xi_{in}, \Theta^{LQ})}_{\Psi', \Delta, \Theta_{comm}} LP'_{k+1}$, see above.
  6. [BisimLQ] Since δ is valid for (Ψ, M) and, by Lemma B.2.11, $a \in \mathsf{mayMove}(LP_k)$, i.e. $a \in M_P \cup \mathsf{mov}(\Theta^{LP})$. This implies, by Lemma B.6.1,

$$LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Xi_{in}, \Theta^{LP})}_{\Psi', \Delta, \Theta_{comm}} LQ'_k.$$

Thus RHS translocating strong simulates LHS in this case, (Similarly for the case where $a \in \mathsf{dom}(\Delta_{ex})$).

Case $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP_k \xrightarrow{@_a \textbf{migrate to } s} LP_{k+1}$ **and** $a \in \mathsf{dom}(\Theta^{LP}_{comm})$**:** This implies:

- $\Psi\delta \Vdash \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP_k|LQ_k) \xrightarrow{\tau} \texttt{new}\ \Delta, (\Theta_{comm} \oplus a \mapsto s)\ \texttt{in}\ (LP_{k+1}|LQ_k)$, by (LTS-BOUND-MIG), (LTS-NEW) and (LTS-PRL).

- By [BisimLP], there exists LP'_{k+1} such that, $\Psi\delta, \Delta, \Theta_{comm} \Vdash LP'_k \xrightarrow{@_a \mathsf{migrate\ to}\ s} LP'_{k+1}$ (†), and $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Theta_{in}, \Theta^{LQ})}_{\Psi, \Delta, (\Theta_{comm} \oplus a \mapsto s)} LP'_{k+1}$ by the definition of translocating strong bisim.

- $\Psi\delta \Vdash \texttt{new}\ \Delta, \Theta_{comm}\ \texttt{in}\ (LP'_k|LQ'_k) \xrightarrow{\tau} \texttt{new}\ \Delta, (\Theta_{comm} \oplus a \mapsto s)\ \texttt{in}\ (LP'_{k+1}|LQ'_k)$ (†), by (LTS-BOUND-MIG), (LTS-NEW) and (LTS-PRL).

- Let $\Psi' = \Psi\delta$, we have:

  $$\texttt{new}\ \Delta, (\Theta_{comm} \oplus a \mapsto s)\ \texttt{in}\ (LP_{k+1}|LQ_k)\ \mathcal{R}^M_{\Psi'}\ \texttt{new}\ \Delta, (\Theta_{comm} \oplus a \mapsto s)\ \texttt{in}\ (LP'_{k+1}|LQ_k)$$

  as there exists

  $$\begin{array}{lcllcl} \Delta'_{ex} & \equiv & \Delta_{ex}\delta & \Xi_{in} & \equiv & \Theta_{in}\delta \\ \Xi^{LP}_{comm} & \equiv & \Theta^{LP}_{comm} \oplus a \mapsto s & \Xi^{LQ}_{comm} & \equiv & \Theta^{LQ}_{comm} \\ \Delta' & \equiv & \Delta & & & \end{array}$$

  and $\Theta^{LP}_{out}, \Theta^{LQ}_{out}$ such that

  1. [TC] $\Psi' \equiv (\Gamma, \Delta_{ex}, \Theta_{in}, \Theta_{out})\delta \equiv \Gamma\delta, \Delta'_{ex}, \Xi_{in}, \Theta_{out}$.
  2. [TL] $M = (\mathsf{mov}(\Gamma) \cap M_P \cap M_Q) \cup \mathsf{mov}(\Xi_{in})$, as $\mathsf{mov}(\Xi_{in}) = \mathsf{mov}(\Theta_{in})$.
  3. [MovLP] $\mathsf{mayMove}(LQ_k, LQ'_k) \subseteq M_P \cup \mathsf{mov}(\Xi^{LQ})$, since $\mathsf{dom}(\Xi^{LQ}) = \mathsf{dom}(\Theta^{LQ})$.
  4. [MovLQ] $\mathsf{mayMove}(LP_{k+1}, LP'_{k+1}) \subseteq M_Q \cup \mathsf{mov}(\Xi^{LQ})$, by Lemma B.2.12.
  5. [BisimLP] $LP_{k+1} \dot{\sim}^{M_P \cup \mathsf{mov}(\Xi_{in}, \Xi^{LQ})}_{\Psi', \Delta, \Xi_{comm}} LP'_{k+1}$, see above.
  6. [BisimLQ] Since δ is valid for (Ψ, M) and, by Lemma B.2.11, we have $a \in \mathsf{mayMove}(LP_k)$, i.e. $a \in M_P \cup \mathsf{mov}(\Theta^{LP})$. This implies $LQ_k \dot{\sim}^{M_Q \cup \mathsf{mov}(\Xi_{in}, \Xi^{LP})}_{\Psi', \Delta, \Xi_{comm}} LQ'_k$, by Lemma B.6.1.

Thus RHS translocating strong simulates LHS in this case, (and similarly for the cases where $a \in \mathsf{dom}(\Delta, \Theta^{LQ}_{comm})$).

Therefore $\mathcal{R}$ is a strong translocation bisimulation.

Replacing all $\dot{\sim}$ with $\dot{\approx}$ and all $\xrightarrow[\Delta]{\beta}$ with $\overset{\hat{\beta}}{\underset{\Delta}{\Longrightarrow}}$ at all (†) in the proof for the strong case and we have a proof for the weak case. ■

B.8 Proof Techniques

Lemma B.8.1 (Injection preserves locality of channels)
If c is a local channel in LP and σ is an injective name substitution then $c\sigma$ is a local channel in $LP\sigma$.

Lemma B.8.2 (Substitution preserves locality of channels)
If c is a local channel in LP and σ is a name substitution with $c \notin \mathsf{range}(\sigma)$ then c is a local channel in $LP\sigma$.

Lemma B.8.3 (Transition preserves locality of channels)
Given that c is a local channel in LP and $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$, if β is not an input label $@_a x\mathbf{?}v$, for some a, x, v, with $c \in \mathsf{fv}(v)$, then c is a local channel in LQ.

Proof: A routine induction on the derivation of $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$. ∎

Lemma B.8.4
Given that c is a local channel and $\Gamma \Vdash LP \xrightarrow[\Xi]{\beta} LQ$ with $c \notin \mathsf{fv}(\beta)$, we have:

- if β is an output label then $\mathsf{readA}(c, LQ) \subseteq \mathsf{readA}(c, LP) \cup \mathsf{agents}(\Xi)$ and $\mathsf{writeA}(c, LQ) \subseteq \mathsf{writeA}(c, LP) \cup \mathsf{agents}(\Xi)$.
- if β is not an output label then $\mathsf{readA}(c, LQ) \subseteq \mathsf{readA}(c, LP)$ and $\mathsf{writeA}(c, LQ) \subseteq \mathsf{writeA}(c, LP)$.

Proof: A routine induction on the derivation of $\Gamma \Vdash LP \xrightarrow[\Xi]{\beta} LQ$. ∎

Lemma B.8.5

1. If c is a local channel in LP and $\Gamma \Vdash LP \xrightarrow[\Xi]{@_a c\mathbf{!}v} LQ$ then $a \in \mathsf{writeA}(c, LP)$.
2. If c is a local channel in LP and $\Gamma \Vdash LP \xrightarrow[\Xi]{@_a c\mathbf{?}v} LQ$ then $a \in \mathsf{readA}(c, LP)$.

Lemma B.8.6 (Scope narrowing of local channel (6.2.7))
Given that Γ is a closed located type context with $\Gamma,\ c : \mathbf{\hat{}}^I T \vdash LP|LQ$, if c is a local channel in LP, LQ and $\mathsf{agents}(LP) \cap \mathsf{agents}(LQ) = \emptyset$ then

$$\mathbf{new}\ c : \mathbf{\hat{}}^I T\ \mathbf{in}\ (LP \mid LQ) \quad \sim_\Gamma \quad (\mathbf{new}\ c : \mathbf{\hat{}}^I T\ \mathbf{in}\ LP) \mid (\mathbf{new}\ c : \mathbf{\hat{}}^I T\ \mathbf{in}\ LQ).$$

Proof: Construct a translocating relation $\mathcal{S}$, with $\mathcal{S}^M_\Phi$ contains the pairs

$$(\mathbf{new}\ c : \mathbf{\hat{}}^I T, \Theta\ \mathbf{in}\ (LP_1 \mid LQ_1),\ \mathbf{new}\ c, c' : \mathbf{\hat{}}^I T, \Theta\ \mathbf{in}\ (LP_1 \mid (\{c'/c\}LQ_1)))$$

subject to the following.

$$\Phi, c : \hat{}^{\mathtt{rw}}T, \Theta \vdash LP_1 \mid LQ_1 \ \wedge\ \vdash \Phi, c : \hat{}^{\mathtt{rw}}T, c' : \hat{}^{\mathtt{rw}}T, \Theta \qquad \text{([Process typing])}$$

$$c \text{ is a local channel in } LP_1, LQ_1 \qquad \text{([Loc])}$$

$$\mathsf{readA}(c, LP_1) \cap \mathsf{writeA}(c, LQ_1) = \mathsf{readA}(c, LQ_1) \cap \mathsf{writeA}(c, LP_1) = \emptyset \qquad \text{([Disj])}$$

We shall show that $\mathcal{S}$ is a translocating strong bisimulation. In fact, since there is no condition on the translocating index, $\mathcal{S}$ is a strong congruence. Once such a result is validated, we may derive the lemma as follows.

$$\begin{array}{rcl}
\mathbf{new}\ c : \hat{}^{I}T\ \mathbf{in}\ (LP \mid LQ) & \sim_\Gamma & \mathbf{new}\ c, c' : \hat{}^{I}T\ \mathbf{in}\ (LP \mid (\{c'/c\}LQ)) \\
& \equiv & (\mathbf{new}\ c : \hat{}^{I}T\ \mathbf{in}\ LP) \mid (\mathbf{new}\ c' : \hat{}^{I}T\ \mathbf{in}\ (\{c'/c\}LQ)) \\
& \equiv & (\mathbf{new}\ c : \hat{}^{I}T\ \mathbf{in}\ LP) \mid (\mathbf{new}\ c : \hat{}^{I}T\ \mathbf{in}\ LQ)
\end{array}$$

Let

$$(\mathbf{new}\ c : \hat{}^{I}T, \Theta\ \mathbf{in}\ (LP_1 \mid LQ_1),\ \mathbf{new}\ c, c' : \hat{}^{I}T, \Theta\ \mathbf{in}\ (LP_1 \mid (\{c'/c\}LQ_1))) \in \mathcal{S}^M_\Phi$$

and δ be a valid relocator for (Φ, M). Supposing that

$$\Phi\delta \Vdash \mathbf{new}\ c : \hat{}^{I}T, \Theta\ \mathbf{in}\ (LP_1 \mid LQ_1) \xrightarrow[\Xi]{\beta} LR$$

We shall demonstrate interesting cases for deriving this transition.

Case $\Phi\delta, c : \hat{}^{\mathtt{rw}}T, \Theta \Vdash LP_1 \xrightarrow[\Xi]{@_a x!v} LP_2$ **where** $x \notin \mathsf{dom}(\Theta, c)$**:** In this case, let $\Theta \equiv \Theta_1, \Theta_2$ with $\mathsf{dom}(\Theta) \cap \mathsf{fv}(v) = \mathsf{dom}(\Theta_1)$; we may derive the following.

- By [Loc], we have c is non-sendable. Applying Lemma 6.2.1, we have $c \notin \mathsf{fv}(v)$.
- $\Phi\delta \Vdash \mathbf{new}\ c : \hat{}^{I}T, \Theta\ \mathbf{in}\ (LP_1 \mid LQ_1) \xrightarrow[\Xi,\Theta_1]{@_a x!v} LR$, by (LTS-OPEN), (LTS-PRL) and (LTS-NEW), where $LR \equiv \mathbf{new}\ c : \hat{}^{I}T, \Theta_2\ \mathbf{in}\ (LP_2 \mid LQ_1)$.
- $\Phi\delta \Vdash \mathbf{new}\ c, c' : \hat{}^{I}T, \Theta\ \mathbf{in}\ (LP_1 \mid (\{c'/c\}LQ_1)) \xrightarrow[\Xi,\Theta_1]{@_a x!v} LR'$, by (LTS-OPEN), (LTS-PRL) and (LTS-NEW), where $LR' \equiv \mathbf{new}\ c, c' : \hat{}^{I}T, \Theta_2\ \mathbf{in}\ (LP_2 \mid (\{c'/c\}LQ_1))$.
- $LR\ \mathcal{S}^M_{\Phi\delta,\Xi,\Theta_1}\ LR'$ since the followings hold:
  - [Process typing]: $\Phi\delta, c : \hat{}^{I}T, \Theta_2, \Xi, \Theta_1 \vdash LP_2$, by Theorem 4.5.1 (Subj).
  - [Loc]: c is a local channel in LP_2, by Lemma B.8.3.
  - [Disj]: $\mathsf{readA}(c, LP_2) \subseteq \mathsf{readA}(c, LP_1)$ and $\mathsf{writeA}(c, LP_2) \subseteq \mathsf{writeA}(c, LP_1)$, by Lemma B.8.4. This implies $\mathsf{readA}(c, LP_2) \cap \mathsf{writeA}(c, LQ_1) = \mathsf{readA}(c, LQ_1) \cap \mathsf{readA}(c, LP_2) = \emptyset$.

Thus, $\mathcal{S}$ is a translocating bisimulation in this case.

Case $\Phi\delta, c : \texttt{\^{}rw}T, \Theta \Vdash LQ_1 \xrightarrow[\Xi]{@_a x\texttt{!}v} LQ_2$ **where** $x \notin \mathsf{dom}(\Theta, c)$**:** Similar to the above case, except we need to show that

$$\Phi\delta \Vdash \textbf{new}\ c, c' : \texttt{\^{}}^I T, \Theta\ \textbf{in}\ (LP_1 \mid (\{c'/c\}LQ_1)) \xrightarrow[\Xi,\Theta_1]{@_a x\texttt{!}v} LR'.$$

This transition can be derived as follows.

- $\Phi\delta, c, c' : \texttt{\^{}rw}T, \Theta \Vdash \{c'/c\}LQ_1 \xrightarrow[\Xi]{@_a x\texttt{!}v} \{c'/c\}LQ_2$, by Lemma 4.5.7 (Lts Inj), and $c \notin \mathsf{fv}(v), x, a$ (then apply Lemma 4.5.5 (Lts SW)).

Thus true for this case.

Case $\Phi\delta, c : \texttt{\^{}rw}T, \Theta \Vdash LP_1 \xrightarrow[\Xi]{@_a x\texttt{?}v} LP_2$ **and** $\Phi\delta, c : \texttt{\^{}rw}T, \Theta \Vdash LQ_1 \xrightarrow[\Xi]{@_a x\texttt{?}v} LQ_2$**:** In this case, we may derive the following.

- $x \neq c$, since $a \in \mathsf{writeA}(c, LP_1) \cap \mathsf{readA}(c, LQ_1)$, by Lemma B.8.5. By [Loc], applying Lemma 6.2.1, $c \notin \mathsf{fv}(v)$.

- $\Phi\delta \Vdash \textbf{new}\ c : \texttt{\^{}}^I T, \Theta\ \textbf{in}\ (LP_1 \mid LQ_1) \xrightarrow{\tau} LR$, by (Lts-Comm) and (Lts-New), where $LR \equiv \textbf{new}\ c : \texttt{\^{}}^I T, \Theta, \Xi\ \textbf{in}\ (LP_2 \mid LQ_2)$.

- $\Phi\delta, c' : \texttt{\^{}rw}T, \Theta \Vdash \{c'/c\}LQ_1 \xrightarrow[\Xi]{@_a x\texttt{?}v} \{c'/c\}LQ_2$, by Lemma 4.5.7 (Lts Inj), and $c \notin \mathsf{fv}(v), x, a$.

- $\Phi\delta \Vdash \textbf{new}\ c, c' : \texttt{\^{}}^I T, \Theta\ \textbf{in}\ (LP_1 \mid (\{c'/c\}LQ_1)) \xrightarrow{\tau} LR'$, by (Lts-Comm) and (Lts-New), where $LR' \equiv \textbf{new}\ c, c' : \texttt{\^{}}^I T, \Theta, \Xi\ \textbf{in}\ (LP_2 \mid (\{c'/c\}LQ_2))$.

- $LR\ \mathcal{S}^M_{\Phi\delta}\ LR'$ since the following hold:
  - [Process typing]: $\Phi\delta, c : \texttt{\^{}}^I T, \Theta, \Xi \vdash LP_2$, by Theorem 4.5.1 (Subj).
  - [Loc]: c is a local channel in LP_2, by Lemma B.8.3.
  - [Disj]: $\mathsf{readA}(c, LP_2) \subseteq \mathsf{readA}(c, LP_1)$ and $\mathsf{writeA}(c, LP_2) \subseteq \mathsf{writeA}(c, LP_1)$, etc. by Lemma B.8.4.
    This implies $\mathsf{readA}(c, LP_2) \cap \mathsf{writeA}(c, LQ_2) = \mathsf{readA}(c, LQ_2) \cap \mathsf{readA}(c, LP_2) = \emptyset$.

Thus, $\mathcal{S}$ is a translocating strong bisimulation. ■

Lemma B.8.7 (Relocatable deterministic reduction)
If $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$ then, for any valid relocator δ for (Γ, M), $\Gamma\delta \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$.

Lemma B.8.8 (Strengthening/weakening deterministic reduction)
Given that $\vdash_{\mathsf{L}} \Gamma, \Delta$ with Δ being extensible and $\Gamma \vdash LP$, $M \subseteq \mathsf{agents}(\Gamma)$, $M' \subseteq \mathsf{agents}(\Delta)$, we have: $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$ if and only if $\Gamma, \Delta \Vdash LP \xrightarrow[M \cup M']{\mathsf{det}} LQ$.

Proof: For the right-to-left direction, supposing $\Gamma, \Delta \Vdash LP \xrightarrow[M \cup M']{\mathsf{det}} LQ$ and $\Gamma \vdash LP$, we may derive the following.

- Let δ be a valid relocator for $((\Gamma, \Delta), M \cup M')$, $(\Gamma, \Delta)\delta \Vdash LP \xrightarrow{\tau} LQ$, by the definition of determinacy. This means that $\Gamma\delta \Vdash LP \xrightarrow{\tau} LQ$, by Lemma 4.5.5 (Lts SW).
- Let δ' be a valid relocator for (Γ, M); Assuming $\Gamma\delta' \Vdash LP \xrightarrow[\Xi]{\beta} LR$, by Lemma 4.5.5 (Lts SW), we have:

  $$\Gamma\delta', \Delta \Vdash LP \xrightarrow[\Xi]{\beta} LR$$

  However, by the definition of determinacy, we have $\beta = \tau$ and $LQ \dot{\sim}^{M \cup M'}_{\Gamma, \Delta} LR$. Applying Lemma 5.3.5 (Transloc Bisim SW), we have: $LQ \dot{\sim}^{M}_{\Gamma} LR$.

Hence $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$, ie. true for this case.

For the left-to-right direction is similar. Omitted. ∎

B.9 Δ-Restricted Equivalences

Names in Φ_{aux} are binding in $\mathcal{C}\,[\![LP]\!]$ and are subject to alpha-conversion. This means that in principle $\mathcal{C}\,[\![LP]\!]$ should be able to extrude, by input or output actions, names which clash with Φ_{aux}. However, the definition of the compositional encoding, $[\![]\!]$, elides Φ_{aux}, making it impossible to explicitly rename Φ_{aux} in the $\mathcal{C}$-translation. We need to deal with situations where an extruded context clashes with Φ_{aux} *without* having to explicitly rename the binding context Φ_{aux}. For this we formulate a concept of Φ_{aux}-restriction on operational equivalences and preorders. In this section, we only give the definition and result for translocating strong bisimulation, but it should be clear that such definitions and results extend naturally to translocating weak bisimulation, translocating expansion, and translocating expansion up to expansion.

Definition B.9.1 (Translocating Δ-Restricted Strong Bisimulation)
A symmetric translocating indexed relation $\mathcal{S}$ is a *translocating Δ-restricted strong bisimulation* if Δ is extensible and, for all closed type context Γ with $\vdash \Gamma, \Delta$ and $M \subseteq \mathsf{mov}(\Gamma)$, $(LP, LQ) \in \mathcal{S}^{M}_{\Gamma}$ implies

- $\Gamma \vdash LP$ and $\Gamma \vdash LQ$,
- for any relocator δ valid for (Γ, M), if $\Gamma\delta \Vdash LP \xrightarrow[\Theta]{\beta} LP'$ and $\mathsf{fv}(\beta) \cap \mathsf{dom}(\Delta) = \emptyset$ then there exists LQ' such that $\Gamma\delta \Vdash LQ \xrightarrow[\Theta]{\beta} LQ'$ and $(LP', LQ') \in \mathcal{S}^{M \uplus_\beta \mathsf{mov}(\Theta)}_{\Gamma\delta\beta,\Theta}$.

Lemma B.9.1
Given that $\mathcal{R}$ is a strong translocating Δ-restricted bisimulation, if $LP\mathcal{R}^M_\Gamma LQ$ then $LP \dot{\sim}^M_\Gamma LQ$.

Proof: Constructing a translocating relation $\mathcal{S}$, such that $\mathcal{S}^{M'}_\Phi$ is as follows.

$$\{(LP', LQ') \mid \exists \sigma \text{ injective } . \, \sigma LP' \; \mathcal{R}^{\sigma M'}_{\sigma\Phi} \; \sigma LQ' \wedge \; \vdash \sigma\Phi, \Delta\}$$

We shall prove that $\mathcal{S}$ is a translocating strong bisimulation.

Supposing $(LP', LQ') \in \mathcal{S}^{M'}_\Phi$, this means that there exists an injection σ such that $\vdash \sigma\Phi, \Delta$ and $\sigma LP' \; \mathcal{R}^{\sigma M'}_{\sigma\Phi} \; \sigma LQ'$. Let δ be a valid relocator for (Φ, M'), and $\Phi\delta \Vdash LP' \xrightarrow[\Theta]{\beta} LP'_1$, we may derive the following.

- Picking an injective $\rho : \mathsf{dom}(\Theta) \to \mathcal{X}/\mathsf{dom}(\Phi, \Delta)$, clearly $\sigma + \rho$ is an injective. By Lemma 4.5.7 (Lts Inj), we have: $\sigma(\Phi\delta) \Vdash \sigma LP' \xrightarrow[\rho\Theta]{(\sigma+\rho)\beta} (\sigma + \rho)LP'_1$.
- Since $\mathsf{dom}(\Theta\rho) \cap \mathsf{dom}(\Delta) = \emptyset$, there exists LQ'_1 such that $\sigma(\Phi\delta) \Vdash \sigma LQ' \xrightarrow[\rho\Theta]{(\sigma+\rho)\beta} LQ'_1$ and $(\sigma + \rho)LP'_1 \; \mathcal{R}^{(\sigma+\rho)(M' \uplus_\beta \mathsf{mov}(\Theta))}_{(\sigma+\rho)(\Phi\delta\beta,\Theta)} \; LQ'_1$, since $\sigma LP' \mathcal{R}^{\sigma M'}_{\sigma\Phi} \sigma LQ'$.
- Since $(\sigma + \rho)$ is an injective, we have $(\sigma + \rho)^{-1}$ is also an injective. By Lemma 4.5.7 (Lts Inj), we have: $\Phi\delta \Vdash LQ' \xrightarrow[\Theta]{\beta} (\sigma + \rho)^{-1}LQ'_1$.
- This implies $LP'_1 \mathcal{S}^{M' \uplus_\beta \mathsf{mov}(\Theta)}_{\Phi\delta,\Theta} (\sigma + \rho)^{-1}LQ'_1$.

This implies $\mathcal{S}$ is a translocating strong simulation. The proof of $\mathcal{S}^{-1}$ being a translocating strong simulation is similar. Hence the lemma. ∎

Appendix C

Properties of Temporary Immobility

Lemma C.0.2 (Blocking set up to $\dot{\approx}$ implies temporary immobility (6.4.6)) Given that Γ is a closed located type context, and $M \subseteq \mathsf{agents}(\Gamma)$ is a translocating index, if there exists a blocking set up to $\dot{\approx}$ under l, $\mathcal{M}$, such that $LP \in \mathcal{M}_{\Gamma}^{M}$ then LP is temporary immobile under l w.r.t. (Γ, M).

Proof: Supposing $\mathcal{M}$ is a blocking set up to $\dot{\approx}$ under l, we construct a translocating set $\mathcal{N}$, defined for any closed located type context Φ, and translocating index M', as follows:

$$\mathcal{N}_{\Phi}^{M'} \stackrel{\text{def}}{=} \{LQ \mid \exists LR \,.\, LQ \dot{\approx}_{\Phi}^{M'} LR \ \wedge\ LR \in \mathcal{M}_{\Phi}^{M'}\}$$

We shall prove that $\mathcal{N}$ is a blocking set under l. Let $LQ \in \mathcal{N}_{\Phi}^{M'}$. By definition there exists LR such that $LQ \dot{\approx}_{\Phi}^{M'} LR$ and $LR \in \mathcal{M}_{\Phi}^{M'}$. Suppose that δ ia a valid relocator for (Φ, M') and $\Phi\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$. The following are all the possibilities. We omit the trivial $\beta = \tau$ case.

Case $\beta = @_b c\,?\,v$ with $l \notin \mathsf{fv}(\beta)$: There exists LR' such that $\Phi\delta \Vdash LR \stackrel{\beta}{\underset{\Delta}{\Longrightarrow}} LR'$ and $LQ' \dot{\approx}_{\Phi\delta,\Delta}^{M' \cup \mathsf{mov}(\Delta)} LR'$, by the definition of translocating weak bisimulation. $LR' \in \mathcal{M}_{\Phi\delta,\Delta}^{M' \cup \mathsf{mov}(\Delta)}$ since $\mathcal{M}$ is a blocking set under l. $LQ' \in \mathcal{N}_{\Phi\delta,\Delta}^{M' \cup \mathsf{mov}(\Delta)}$, by definition.

Case $\beta = @_b c\,!\,v$: There exists LR' such that $\Phi\delta \Vdash LR \stackrel{\beta}{\underset{\Delta}{\Longrightarrow}} LR'$ and $LQ' \dot{\approx}_{\Phi\delta,\Delta}^{M'} LR'$, by the definition of translocating weak bisimulation. $LR' \in \mathcal{M}_{\Phi\delta,\Delta}^{M'}$ and $l \notin \mathsf{fv}(\beta)$, since $\mathcal{M}$ is a blocking set. $LQ' \in \mathcal{N}_{\Phi\delta,\Delta}^{M'}$, by definition.

Case $\beta = @_b\textsf{migrate to } s$**:** There exists LR' such that $\Phi\delta \Vdash LR \overset{@_b\textsf{migrate to } s}{\Longrightarrow} LR'$ and $LQ' \dot{\approx}^{M'}_{\Phi\delta @_b\textsf{migrate to } s} LR'$, by the definition of translocating weak bisimulation. However, $\mathcal{M}$ is a blocking set implies that such an LR' does not exist. A contradiction occurs. Therefore b, s, LQ' do not exist.

Hence the lemma. ■

Lemma C.0.3 (Safety of deterministic reduction (6.4.2))
Given that LQ is temporary immobile under l w.r.t. $((\Gamma, \Delta), M)$, with Δ being extensible and $l \in \mathsf{dom}(\Delta)$, if $\Gamma, \Delta \Vdash LP_1 \xrightarrow[M]{\mathsf{det}} LP_2$ then

$$\textbf{new } \Delta \textbf{ in } (LP_1 \mid LQ) \stackrel{\cdot}{\succeq}^{M \cap \mathsf{dom}(\Gamma)}_{\Gamma} \textbf{new } \Delta \textbf{ in } (LP_2 \mid LQ)$$

Proof: Construct a translocating relation $\mathcal{R}$ with $\mathcal{R}^{M'}_{\Xi}$ the union of $\equiv$ and the pairs

$$(\textbf{new } \Delta' \textbf{ in } (LP_1 \mid LQ'),\ \textbf{new } \Delta' \textbf{ in } (LP_2 \mid LQ'))$$

subject to the condition that there exist $\Gamma', \Theta_{in}, \Theta_{ex}, M''$, with Γ' closed, and Θ_{in} and Θ_{ex} extensible, satisfying

$$\begin{array}{lr}
\Xi \equiv \Gamma', \Theta_{ex}, \Theta_{in} & ([\text{Cont}]) \\
\Xi \vdash \textbf{new } \Delta' \textbf{ in } (LP_1 | LQ') & ([\text{Type}]) \\
M' = (M'' \cap \mathsf{mov}(\Gamma')) \cup \mathsf{mov}(\Theta_{in}) & ([\text{Transl}]) \\
l \in \mathsf{dom}(\Delta') & ([\text{Lock}]) \\
LQ' \in \mathcal{M}^{M'' \cup \mathsf{mov}(\Theta_{in})}_{\Xi, \Delta'} & ([\text{TempImmob}]) \\
\Xi, \Delta' \Vdash LP_1 \xrightarrow[M'' \cup \mathsf{mov}(\Theta_{ex}, \Theta_{in})]{\mathsf{det}} LP_2 & ([\text{Determ}])
\end{array}$$

We shall prove that $\mathcal{R}$ is a translocating expansion. Let δ be a valid relocator for (Ξ, M') and $(\textbf{new } \Delta' \textbf{ in } (LP_1 \mid LQ'), \textbf{new } \Delta' \textbf{ in } (LP_2 \mid LQ')) \in \mathcal{R}^{M'}_{\Xi}$. By the definition of $\mathcal{R}$, there exists $\Gamma', \Theta_{in}, \Theta_{ex}, M''$ satisfying all the above conditions.

RHS strictly simulates LHS Supposing $\Xi\delta \Vdash \textbf{new } \Delta' \textbf{ in } (LP_1 \mid LQ') \xrightarrow[\Theta]{\beta} LR$, this implies, by Lemma B.2.3 (Lts Analysis), that one of the following cases holds. We omit trivial cases.

Case $\Xi\delta, \Delta' \Vdash LP_1 \xrightarrow{\tau} LP'$**:** Applying the definition of determinism on [Determ], we have $LP' \equiv LP_2$. $\beta = \tau$, $\Theta = \bullet$ and $LR \equiv \textbf{new } \Delta' \textbf{ in } LP_2 \mid LQ'$. As $\equiv \subseteq \mathcal{R}$, we have $(LR, \textbf{new } \Delta' \textbf{ in } (LP_2 \mid LQ')) \in \mathcal{R}^{M'}_{\Xi}$ Thus RHS strictly simulates LHS in this case.

Case $\Xi\delta, \Delta' \Vdash LP_1 \xrightarrow{\beta} LP'$**:** Applying the definition of determinism on [Determ], we have $\beta = \tau$, which is the previous case.

Case $\Xi\delta, \Delta' \Vdash LQ' \xrightarrow[\Theta]{@_a c\textbf{?}v} LQ''$ **with** $\mathsf{dom}(\Delta') \cap \mathsf{fv}(@_a c\textbf{?}v) = \emptyset$**:** In this case, we may derive the following.

- $\beta = @_a c\textbf{?}v$, $\Theta = \Theta$ and $LR \equiv \texttt{new}\ \Delta'\ \texttt{in}\ (LP_1 \mid LQ'')$.
- $\Xi\delta \Vdash \texttt{new}\ \Delta'\ \texttt{in}\ (LP_2 \mid LQ') \xrightarrow[\Theta]{@_a c\textbf{?}v} \texttt{new}\ \Delta'\ \texttt{in}\ (LP_2 \mid LQ'')$, by (Lts-Prl), (Lts-New).
- $(LR, \texttt{new}\ \Delta'\ \texttt{in}\ (LP_2 \mid LQ'')) \in \mathcal{R}^{M' \cup \mathsf{mov}(\Theta)}_{\Xi\delta,\Theta}$ since there exists:

$$\Gamma'' = \Gamma'\delta \qquad \Theta'_{in} = \Theta_{in}\delta, \Theta \qquad \Theta'_{ex} = \Theta_{ex}$$

  such that:

  - [Cont] $\Xi\delta, \Theta = \Gamma'', \Theta'_{in}, \Theta'_{ex}$.
  - [Type] $\Xi\delta, \Theta \vdash LR$, by Theorem 4.5.1 (Subj).
  - [Transl]

$$M'' \cup \mathsf{mov}(\Theta) \quad = (M'' \cap \mathsf{mov}(\Gamma')) \cup \mathsf{mov}(\Theta_{in}, \Theta) \quad = (M'' \cap \mathsf{mov}(\Gamma')) \cup \mathsf{mov}(\Theta'_{in})$$

    Since $M'' \subseteq \mathsf{agents}(\Gamma', \Delta')$ and hence $M'' \cap \mathsf{dom}(\Theta) = \emptyset$.

  - [Lock] $l \in \mathsf{dom}(\Delta')$.
  - [Determ] $\Xi\delta, \Theta, \Delta' \Vdash LP_1 \xrightarrow[M'' \cup \mathsf{mov}(\Theta'_{ex}, \Theta'_{in})]{\mathsf{det}} LP_2$, by Lemma B.8.7 and Lemma B.8.8.
  - [TempImmob] Since $l \notin \mathsf{fv}(v)$, we have: $LQ' \in \mathcal{M}^{M'' \cup \mathsf{mov}(\Theta'_{in})}_{\Xi\delta,\Theta,\Delta'}$, by the closure property of $\mathcal{M}$ being a blocking set.

Thus true for this case.

Case $\Xi\delta, \Delta' \Vdash LQ' \xrightarrow[\Theta]{@_a c\textbf{!}v} LQ''$ **with** $a, c \in \mathsf{dom}(\Xi)$**:** In this case, we may derive the following.

- $l \notin \mathsf{fv}(@_a c\textbf{!}v)$ since $LQ' \in \mathcal{M}^{M'' \cup \mathsf{mov}(\Theta_{in})}$, apply Lemma 6.4.3.
- Assuming $\Delta' = \Delta'_1, \Delta'_2$ where $\mathsf{dom}(\Delta') \cap \mathsf{fv}(v) = \mathsf{dom}(\Delta'_1)$.
- $\beta = @_a c\textbf{!}v$, $\Theta = \Theta, \Delta'_1$ and $LR \equiv \texttt{new}\ \Delta'_2\ \texttt{in}\ (LP_1 \mid LQ'')$.
- $\Xi\delta \Vdash \texttt{new}\ \Delta'\ \texttt{in}\ (LP_2 \mid LQ') \xrightarrow[\Theta, \Delta'_1]{@_a c\textbf{!}v} \texttt{new}\ \Delta'_2\ \texttt{in}\ (LP_2 \mid LQ'')$, by (Lts-Prl), (Lts-New) and (Lts-Open).

- $(LR, \mathbf{new}\ \Delta'\ \mathbf{in}\ (LP_2 \mid LQ'')) \in \mathcal{R}^{M'}_{\Xi\delta,\Theta,\Delta'_1}$ since there exists:

$$\Gamma'' = \Gamma'\delta, \Delta'_1 \qquad \Theta'_{in} = \Theta_{in}\delta \qquad \Theta'_{ex} = \Theta_{ex}, \Theta$$

  such that:

  - [Cont] $\Xi\delta, \Theta, \Delta'_1 = \Gamma'', \Theta'_{in}, \Theta'_{ex}$.
  - [Type] $\Xi\delta, \Theta, \Delta'_1 \vdash LR$, by Theorem 4.5.1 (Subj).
  - [Transl]

$$M' \quad = (M'' \cap \mathsf{mov}(\Gamma')) \cup \mathsf{mov}(\Theta_{in}) \quad = (M'' \cap \mathsf{mov}(\Gamma')) \cup \mathsf{mov}(\Theta'_{in})$$

  - [Lock] $l \notin \mathsf{fv}(v)$ implies $l \in \mathsf{dom}(\Delta'_2)$.
  - [Determ] $\Xi\delta, \Theta, \Delta' \Vdash LP_1 \xrightarrow[M''\cup\mathsf{mov}(\Theta'_{ex},\Theta'_{in})]{\mathsf{det}} LP_2$, by Lemma B.8.7 and Lemma B.8.8.
  - [TempImmob] $LQ' \in \mathcal{M}^{M''\cup\mathsf{mov}(\Theta'_{in})}_{\Xi\delta,\Theta,\Delta'}$, by the closure property of $\mathcal{M}$ being a blocking set.

Thus true for this case.

Case Communication between LP_1 and LQ': Impossible since LP_1 can neither send nor receive.

Case $\Xi\delta, \Delta' \Vdash LQ' \xrightarrow{@_a\mathsf{migrate\ to}\ s} LQ''$: Impossible, by [TempImmob].

LHS progressingly simulates RHS Whatever action committed by RHS, LHS can do one silent action and commit the same action. ■

Lemma C.0.4 (Immobility)
Given that LP is a located process such that $\mathsf{mayMove}(LP) = \emptyset$, if Γ is a closed located type context with $\Gamma \vdash P$, $M \subseteq \mathsf{mov}(\Gamma)$ and l is non-sendable in LP, then LP is temporary immobile under l w.r.t. (Γ, M).

Lemma C.0.5 (Composition preserve temporary immobility (6.4.7))
If LP and LQ are temporarily immobile under l, w.r.t. $((\Gamma, \Delta), M)$ with $l \in \mathsf{dom}(\Gamma)$ and Δ extensible, then $\mathbf{new}\ \Delta\ \mathbf{in}\ (LP|LQ)$ is temporarily immobile under l, w.r.t. $(\Gamma, M \cap \mathsf{dom}(\Gamma))$.

Proof: Given that LP, LQ are temporarily immobile under l, wrt $((\Gamma, \Delta), M)$, there exist blocking sets under l, $\mathcal{M}, \mathcal{N}$ such that $LP \in \mathcal{M}^M_{\Gamma,\Delta}$ and $LQ \in \mathcal{N}^M_{\Gamma,\Delta}$. We choose those which satisfy the following.

- They are translocating. $LR \in \mathcal{M}_{\Phi}^{M'}$ and δ be a valid relocators for (Φ, M') implies $LR \in \mathcal{M}_{\Phi\delta}^{M'}$.
- They can be weakened. $LR \in \mathcal{M}_{\Phi}^{M'}$, $\vdash \Phi, \Theta$ and $M'' \subseteq \mathsf{mov}(\Theta)$ implies $LR \in \mathcal{M}_{\Phi,\Theta}^{M' \cup M''}$.

It is easy to prove that choosing blocking sets which satisfy the above is always possible.

We construct a translocating set $\mathcal{O}$, such that

$$\mathcal{O}_{\Phi}^{M'} \stackrel{\text{def}}{=} \left\{ \texttt{new}\ \Theta\ \texttt{in}\ (LP'|LQ') \;\middle|\; \exists M_P, M_Q\ .\ \begin{array}{l} M' \subseteq (M_P \cap M_Q \cap \mathsf{dom}(\Phi)) \wedge \\ LP' \in \mathcal{M}_{\Phi,\Theta}^{M_P} \;\wedge\; LQ' \in \mathcal{N}_{\Phi,\Theta}^{M_Q} \end{array} \right\}$$

We need to prove that $\mathcal{O}$ is a blocking set under l. Let δ be a valid relocator for (Φ, M') and consider $\texttt{new}\ \Theta\ \texttt{in}\ (LP'|LQ') \in \mathcal{O}_{\Phi}^{M'}$. By definition of $\mathcal{O}$, there exists M_P, M_Q such that the above holds. Suppose $\Phi\delta \Vdash \texttt{new}\ \Theta\ \texttt{in}\ (LP'|LQ') \xrightarrow[\Xi]{\beta} LR$, one of the following cases holds.

Case $\beta = \tau$: By Lemma B.2.3 (Lts Analysis), these are all the possibilities (we ignore symmetric cases).

- $\Phi\delta, \Theta \Vdash LP' \xrightarrow{\tau} LP''$ and $LR \equiv \texttt{new}\ \Theta\ \texttt{in}\ (LP''|LQ')$. $LP' \in \mathcal{M}_{\Phi,\Theta}^{M_P}$ implies $LP'' \in \mathcal{M}_{\Phi\delta,\Theta}^{M_P}$. $LQ' \in \mathcal{N}_{\Phi\delta,\Theta}^{M_Q}$, since $\mathcal{N}$ satisfies "translocating" condition above.
- $\Phi\delta, \Theta \Vdash LP' \xrightarrow[\Xi]{@_a c?v} LP''$, $\Phi\delta, \Theta \Vdash LQ' \xrightarrow[\Xi]{@_a c!v} LQ''$ and $LR \equiv \texttt{new}\ \Theta, \Xi\ \texttt{in}\ LP''|LQ''$. $LQ' \in \mathcal{M}_{\Phi,\Theta}^{M_Q}$ implies $l \notin \mathsf{fv}(c, v)$ and $LQ'' \in \mathcal{M}_{\Phi\delta,\Theta,\Xi}^{M_Q}$. $LP' \in \mathcal{M}_{\Phi,\Theta}^{M_P}$ and $l \notin \mathsf{fv}(c, v)$ implies $LP'' \in \mathcal{M}_{\Phi\delta,\Theta,\Xi}^{M_P \cup \mathsf{mov}(\Xi)}$.

This means that $LR \in \mathcal{O}_{\Phi\delta}^{M'}$. Thus true for this case.

Case $\beta = @_a c!v$: By Lemma B.2.3 (Lts Analysis), ignoring symmetric cases, we have $\Phi\delta, \Theta \Vdash LP' \xrightarrow[\Xi']{@_a c!v} LP''$. Supposing $\mathsf{fv}(v) \cap \mathsf{dom}(\Theta) = \mathsf{dom}(\Theta_1)$ where $\Theta \equiv \Theta_1, \Theta_2$, by (Lts-Open) etc, we have $LR \equiv \texttt{new}\ \Theta_2\ \texttt{in}\ LP''|LQ'$ and $\Xi = \Xi', \Theta_1$. $LP' \in \mathcal{M}_{\Phi,\Theta}^{M_P}$ implies $l \notin \mathsf{fv}(\beta)$ and $LP'' \in \mathcal{M}_{\Phi\delta,\Theta,\Xi'}^{M_P}$. $LQ' \in \mathcal{N}_{\Phi\delta,\Theta,\Xi'}^{M_Q}$, since $\mathcal{N}$ satisfies the "weakening" condition above. This means that $LR \in \mathcal{O}_{\Phi\delta,\Xi',\Theta_1}^{M'}$. Thus true for this case.

Case $\beta = @_a c?v$ with $l \notin \mathsf{fv}(\beta)$: By Lemma B.2.3 (Lts Analysis), ignoring symmetric cases, we have $\Phi\delta, \Theta \Vdash LP' \xrightarrow[\Xi]{@_a c?v} LP''$ and $a, c \in \mathsf{dom}(\Phi)$. Pick a injection $\sigma : \mathsf{dom}(\Xi) \to \mathcal{X}/\mathsf{dom}(\Phi, \Theta)$, by Lemma 4.5.7 (Lts Inj) $\Phi\delta, \Theta \Vdash LP' \xrightarrow[\sigma\Xi]{@_a c?\sigma v} LP'''$, where $LP''' = \sigma LP''$. By (Lts-Prl) etc, we have $LR \equiv \texttt{new}\ \Theta\ \texttt{in}\ LP'''|LQ'$. $LP' \in \mathcal{M}_{\Phi,\Theta}^{M_P}$ and $l \notin \mathsf{fv}(\beta)$ implies $LP'' \in \mathcal{M}_{\Phi\delta,\Theta,\sigma\Xi}^{M_P \cup \mathsf{mov}(\sigma\Xi)}$. $LQ' \in \mathcal{N}_{\Phi\delta,\Theta,\sigma\Xi}^{M_Q \cup \mathsf{mov}(\sigma\Xi)}$, since $\mathcal{N}$ satisfies the "translocating" and "weakening" conditions above. This means that $LR \in \mathcal{O}_{\Phi,\sigma\Xi}^{M' \cup \mathsf{mov}(\sigma\Xi)}$. Thus true for this case.

Case $\beta = @_a$migrate to s, $a \in \mathsf{dom}(\Phi)$: By Lemma B.2.3 (Lts Analysis), ignoring symmetric cases, we have $\Phi\delta, \Theta \Vdash LP' \xrightarrow{@_a\mathsf{migrate\ to}\ s} LP''$. This is impossible, however, since $LP' \in \mathcal{M}^{M_P}_{\Phi,\Theta}$. A contradiction has occurred. Thus this transition is impossible. ∎

Lemma C.0.6 (Weak bisimulation preserves temporary immobility (6.4.8))
If LP is temporary immobile under l w.r.t. (Γ, M) and $LP \dot{\approx}^M_\Gamma LQ$ then LQ is temporary immobile under l w.r.t. (Γ, M).

Proof: Quite obvious from the proof of Lemma 6.4.6. ∎

C.1 Proofs of $\mathcal{C}$-Related Results

Lemma C.1.1 (Blocked by local lock (6.4.4))
Given that Γ is a closed located type context, $\Gamma \vdash_a P$, $\vdash \Gamma, \Phi_{aux}$ and $M \subseteq \mathsf{agents}(\Gamma)/\{a\}$. $@_a(\llbracket P \rrbracket_a | \mathtt{Deliverer})$ is temporarily immobile under $\mathtt{currentloc}$ w.r.t. $((\Gamma, \Phi_{aux}), M)$.

Proof: We need to consider a translocating derivative of $LP = @_a(\llbracket P \rrbracket_a | \mathtt{Deliverer})$, which is in the form of

$$\mathbf{new}\ \Theta\ \mathbf{in}\ (\prod_i @_{a_i} c_i \,!\, v_i \mid @_a(\llbracket R \rrbracket_a | \mathtt{Deliverer}))$$

where $\prod_i @_{a_i} c_i \,!\, v_i$ are inter-agent messages produced, with Θ being the potentially extruded context. We define a translocating set $\mathcal{M}$ such that $\mathcal{M}^{M'}_{\Psi,\Phi_{aux}}$ is a set of processes of the form

$$\mathbf{new}\ \Theta\ \mathbf{in}\ (\prod_i @_{a_i} c_i \,!\, v_i | \ @_a(\llbracket R \rrbracket_a | \mathtt{Deliverer}))$$

well-typed wrt Ψ, Φ_{aux}, satisfying the following.

$$M' \subseteq \mathsf{agents}(\Psi)/\{a\} \qquad ([\text{Transloc}])$$
$$\mathtt{currentloc} \notin \mathsf{fv}(R, \prod_i @_{a_i} c_i \,!\, v_i) \qquad ([\text{FreeV}])$$

We shall prove that $\mathcal{M}$ is a blocking set up to $\dot{\approx}$ under $\mathtt{currentloc}$.

Let δ be a valid relocator for $((\Psi, \Phi_{aux}), M')$ and $LP \in \mathcal{M}^{M'}_{\Psi,\Phi_{aux}}$, where

$$LP \ \equiv\ \mathbf{new}\ \Theta\ \mathbf{in}\ (\prod_i @_{a_i} c_i \,!\, v_i | \ @_a(\llbracket R \rrbracket_a \ | \ \mathtt{Deliverer}))$$

Supposing $\Psi\delta, \Phi_{aux} \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ with β not being an input label with $\mathtt{currentloc} \in \mathsf{fv}(\beta)$, we shall check all the possibilities that this transition may occur (omitting trivial and similar cases).

Case $R \equiv P'|Q$ and P' is a `create` or `migrate to` process: The channel `currentloc` must be acquired in the first step. Thus true for this case.

Case $R \equiv P'|Q$ and $P' = \langle b@?\rangle c!v$: In this case, we have:

- By the definition of $[\![]\!]_a$, we have $[\![P']\!]_a \stackrel{\text{def}}{=} \langle D@SD\rangle \texttt{message!}\ \{T\}\ [b\ c\ v]$, where $\Psi, \Phi_{aux}, \Theta \vdash v \in T$.
- We decompose LP as $\mathcal{E}[LQ]$ (and clearly $LP \equiv \mathcal{E}[LQ]$) where

$$
\begin{array}{rcl}
\mathcal{E}[\cdot] & = & \textbf{new}\ \Theta\ \textbf{in}\ [\cdot] \mid @_a([\![Q]\!]_a | \texttt{Deliverer}) \\
LQ & = & \prod_i @_{b_i} c_i ! v_i \mid @_a [\![P']\!]_a
\end{array}
$$

- By Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), $@_a\ [\![P']\!]_a \succeq_{\mathcal{E}[\Psi,\Phi_{aux}]} @_D\texttt{message!}\ \{T\}\ [b\ c\ v]$, since D is a static agent in Φ_{aux} (ie. $D \notin \mathsf{mov}(\mathcal{E}[\Psi, \Phi_{aux}])$).
- Applying Theorem 5.4.4 (Transloc Bisim Cong), we have $LQ \succeq_{\mathcal{E}[\Psi,\Phi_{aux}]} LQ_1$, where

$$
LQ_1 \quad = \quad \prod_i @_{b_i} c_i ! v_i \mid @_D\texttt{message!}\ \{T\}\ [b\ c\ v]
$$

  Also, $LP \dot{\succeq}^{M'}_{\Psi,\Phi_{aux}} \mathcal{E}[LQ_1]$, by Theorem 5.4.3 (cong property for expansion).
- $\mathcal{E}[LQ_1] \in \mathcal{M}^{M'}_{\Psi\delta,\Phi_{aux}}$, since $\mathcal{M}$ is a blocking set up to $\dot{\approx}$ under l.

Thus true for this case.

Case $R \equiv P'|Q$ and $P' =$ `iflocal` $\langle b\rangle c!v$ `in` P_1 `else` P_2 with $\Psi\delta, \Phi_{aux} \vdash a@s \wedge b@s$: In this case, we have:

- By the definition of $[\![]\!]$, we have $[\![P']\!]_a \stackrel{\text{def}}{=} \textbf{iflocal}\ \langle b\rangle c!v\ \textbf{in}\ [\![P_1]\!]_a\ \textbf{else}\ [\![P_2]\!]_a$.
- We decompose LP as $\mathcal{E}[LQ]$ (and clearly $LP \equiv \mathcal{E}[LQ]$) where

$$
\begin{array}{rcl}
\mathcal{E}[\cdot] & = & \textbf{new}\ \Theta\ \textbf{in}\ (\prod_i @_{b_i} c_i ! v_i \mid [\cdot] \mid @_a([\![Q]\!]_a | \texttt{Deliverer})) \\
LQ & = & @_a [\![P']\!]_a
\end{array}
$$

- By (LTS-L-IFLOCAL-TRUE), (LTS-PRL) and (LTS-NEW), we have $\Psi\delta, \Phi_{aux} \Vdash LP \xrightarrow{\tau} LP'$ where $LP' = \mathcal{E}[LQ']$ with $LQ' \equiv @_b c!v \mid @_a\ [\![P_1]\!]_a$.
- $LP' \in \mathcal{M}^{M'}_{\Psi\delta,\Phi_{aux}}$, since $\Psi\delta, \Phi_{aux} \vdash LP'$ (by Theorem 4.5.1 (Subj)) and the following holds:

- [Transloc] $M' \subseteq \mathsf{agents}(\Psi\delta)/\{a\}$.
- [FreeV] $\texttt{currentloc} \notin \mathsf{fv}(R)$ implies $\texttt{currentloc} \notin \mathsf{fv}(Q, @_b c\,\texttt{!}\,v)$.

Thus true for this case.

Case $R \equiv P'|Q$ and $P' = c\texttt{!}v|c\texttt{?}p \rightarrow P''$: Similar to the previous case. One may easily observe that, by [FreeV], $\texttt{currentloc}$ is not involved in this communication (ie. $\texttt{currentloc} \notin \mathsf{fv}(c\texttt{!}v)$).

Case $R \equiv P'|Q$ and $P' = c\texttt{?}p \rightarrow P''$: In this case, we have:

- By the definition of $[\![]\!]$, we have $[\![P']\!]_a \stackrel{\text{def}}{=} c\texttt{?}p \rightarrow [\![P'']\!]_a$.
- We decompose LP as $\mathcal{E}[LQ]$ (and clearly $LP \equiv \mathcal{E}[LQ]$) where

$$\begin{array}{rcl} \mathcal{E}[\cdot] & = & \textbf{new}\ \Theta\ \textbf{in}\ (\prod_i @_{b_i} c_i\texttt{!}v_i \mid [\cdot] \mid @_a([\![Q]\!]_a | \texttt{Deliverer})) \\ LQ & = & @_a [\![P']\!]_a \end{array}$$

- By (Lts-L-In), (Lts-Prl) and (Lts-New), we have $\Psi\delta, \Phi_{aux} \Vdash LP \xrightarrow[\Delta]{@_a c\texttt{?}v} LP'$ where $LP' = \mathcal{E}[LQ']$ with

$$LQ' \equiv @_a \mathsf{match}(p, v) [\![P'']\!]_a = @_a [\![\mathsf{match}(p, v) P'']\!]_a$$

  where $\Psi, \Phi_{aux} \vdash c \in \hat{}^{\mathtt{r}} T$, $\Psi, \Phi_{aux}, \Delta \vdash v \in T$ and Δ is an extensible context with $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v)$ and $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Theta) = \emptyset$.

- If $\texttt{currentloc} \notin \mathsf{fv}(@_a c\texttt{?}v)$ then $LP' \in \mathcal{M}^{M' \cup \mathsf{mov}(\Delta)}_{\Psi\delta, \Phi_{aux}, \Delta}$, since $\Psi\delta, \Phi_{aux}, \Delta \vdash LP'$ (by Theorem 4.5.1) and the following holds:
  - [Transloc]: $M' \cup \mathsf{mov}(\Delta) \subseteq \mathsf{agents}(\Psi\delta, \Delta)/\{a\}$, since $\vdash \Psi, \Phi_{aux}, \Delta$.
  - [FreeV]: $\texttt{currentloc} \notin \mathsf{fv}(R, v)$ implies $\texttt{currentloc} \notin \mathsf{fv}(Q, \mathsf{match}(p, v)P'')$.

Thus true for this case.

Case Input via `deliver`: In this case, we have:

- We decompose LP as $\mathcal{E}[LQ]$ (and clearly $LP \equiv \mathcal{E}[LQ]$) where

$$\begin{array}{rcl} \mathcal{E}[\cdot] & = & \textbf{new}\ \Theta\ \textbf{in}\ (\prod_i @_{b_i} c_i\texttt{!}v_i \mid [\cdot] \mid @_a [\![R]\!]_a) \\ LQ & = & @_a\texttt{Deliverer} \end{array}$$

- By (LTS-L-IN), (LTS-PRL) and (LTS-NEW), we have

$$\Psi\delta, \Phi_{aux} \Vdash LP \xrightarrow[\Delta]{@_a\mathtt{deliver?}\{T\}[c\ v\ \mathtt{dack}]} LP'$$

 where $LP' = \mathcal{E}[LQ']$ with

$$LQ' \stackrel{\text{def}}{=} @_a(\mathtt{Deliverer} \mid c\,\mathtt{!}\,v \mid \langle D@SD\rangle \mathtt{dack!}[])$$

 where $\Psi, \Phi_{aux}, \Delta \vdash \{T\}[c\ v\ \mathtt{dack}] \in \{X\}[\mathtt{\hat{~}w}X\ X\ \mathtt{\hat{~}w}[]]$ and Δ is an extensible context with $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(\{T\}[c\ v\ \mathtt{dack}])$ and $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Theta) = \emptyset$.

- By Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), we have $@_a\langle D@SD\rangle\mathtt{dack!}[] \succeq_{\mathcal{E}[\Psi,\Phi_{aux},\Delta]} @_D\mathtt{dack!}[]$, since D is a static agent in the context.

- Applying Theorem 5.4.4 (Transloc Bisim Cong), we have $LQ' \succeq_{\mathcal{E}[\Psi,\Phi_{aux},\Delta]} LQ_1$, where

$$LQ_1 \quad = \quad @_D\mathtt{dack!}[] \mid @_a c\,\mathtt{!}\,v \mid @_a\mathtt{Deliverer}$$

 Also, $LP' \dot{\succeq}^{M'}_{\Psi,\Delta,\Phi_{aux}} \mathcal{E}[LQ_1]$, by Theorem 5.4.3 (cong property for expansion).

- Assuming $\mathtt{currentloc} \notin \mathsf{fv}(@_a\mathtt{deliver?}\{T\}[c\ v\ \mathtt{dack}])$, $LP' \in \mathcal{M}^{M' \cup \mathsf{mov}(\Delta)}_{\Psi\delta,\Phi_{aux},\Delta}$, since $\Psi\delta, \Phi_{aux}, \Delta \vdash LP'$ (by Theorem 4.5.1 (Subj)), $\mathcal{M}$ is a blocking set up to $\dot{\approx}$ under l, and the following holds:
  - [Transloc]: Since Δ is fresh, $a \notin M' \cup \mathsf{mov}(\Delta)$. This implies $M' \cup \mathsf{mov}(\Delta) \subseteq \mathsf{agents}(\Psi\delta, \Delta)/\{a\}$.
  - [FreeV]: $\mathtt{currentloc} \notin \mathsf{fv}(\{T\}[c\ v\ \mathtt{dack}])$ implies $\mathtt{currentloc} \notin \mathsf{fv}(@_a c\,\mathtt{!}\,v)$.

Thus true for this case.

Case $R \equiv P'|Q$ **and** $P' = c\,\mathtt{!}\,v$**:** In this case, we have:

- By the definition of $[\![]\!]$, we have $[\![P']\!]_a \stackrel{\text{def}}{=} c\,\mathtt{!}\,v$.

- Let $\Theta \equiv \Theta_1, \Theta_2$ with $\mathsf{fv}(v) \cap \mathsf{dom}(\Theta) = \mathsf{dom}(\Theta_1)$, by (LTS-L-OUT), (LTS-OPEN), (LTS-PRL) and (LTS-NEW), we have $\Psi\delta, \Phi_{aux} \Vdash LP \xrightarrow[\Theta_1]{@_a c!v} LP'$ where

$$LP' \equiv \mathbf{new}\ \Theta_2\ \mathbf{in}\ (\prod_i @_{b_i} c_i\,\mathtt{!}\,v_i \mid @_a([\![Q]\!]_a \mid \mathtt{Deliverer})).$$

- $\mathtt{currentloc} \notin \mathsf{fv}(P)$ implies (by Lemma B.1.3 (Type FV) $\mathtt{currentloc} \notin \mathsf{fv}(@_a c\,\mathtt{!}\,v)$.

- $LP' \in \mathcal{M}^{M'}_{\Psi\delta,\Phi_{aux},\Theta_1}$, since $\Psi\delta, \Phi_{aux}, \Theta_1 \vdash LP'$ (by Theorem 4.5.1 (Subj)) and the following holds:

– [Transloc]: Since $a \in \mathsf{dom}(\Psi)$, we have $M' \subseteq \mathsf{agents}(\Psi\delta, \Theta_1)/\{a\}$.

– [FreeV]: By Theorem 4.5.1 (Subj), $\mathtt{currentloc} \notin \mathsf{fv}(R)$ implies $\mathtt{currentloc} \notin \mathsf{fv}(Q)$.

Thus true for this case. ■

Lemma C.1.2 (Blocked by acknowledgement (6.4.5))
Let Φ^-_{aux} be defined as follows.

$$\Phi^-_{aux} \stackrel{\text{def}}{=} \Phi_{aux}/\mathtt{currentloc} : \mathtt{\hat{}^{rw}Site}$$

Given that Γ is a closed located type context, $\{\mathtt{ack}, \mathtt{currentloc}\} \cap \mathsf{fv}(P, Q) = \emptyset$, and $\Gamma,\ \Phi^-_{aux},\ \mathtt{ack} : \mathtt{\hat{}^{rw}}T \vdash LP$, where LP is defined below.

$$LP = \mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{}^{rw}Site}\ \mathbf{in}\ @_a([\![P]\!]_a \mid (\mathtt{ack?}p \rightarrow Q) \mid \mathtt{Deliverer})$$

LP is temporarily immobile under $\mathtt{ack}$ w.r.t. $((\Gamma,\ \Phi^-_{aux},\ \mathtt{ack} : \mathtt{\hat{}^{rw}}T), M)$.

Proof: Similar to that of Lemma 6.4.4. Ee define a translocating set $\mathcal{M}$ such that $\mathcal{M}^{M'}_{\Psi,\Phi^-_{aux},\mathtt{ack}:\mathtt{\hat{}^{rw}}T}$ is a set of processes

$$\mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{}^{rw}Site}, \Theta\ \mathbf{in}\ (\prod_i @_{b_i} c_i\mathtt{!}v_i \mid @_a([\![R]\!]_a \mid (\mathtt{ack?}p \rightarrow Q) \mid \mathtt{Deliverer}))$$

well-typed wrt. $\Psi, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T$, satisfying the following.

$$M' \subseteq \mathsf{agents}(\Psi)/\{a\} \qquad \text{([Transloc])}$$
$$\mathtt{ack}, \mathtt{currentloc} \notin \mathsf{fv}(Q, R, \prod_i @_{b_i} c_i\mathtt{!}v_i) \qquad \text{([FreeV])}$$

We shall prove that $\mathcal{M}$ is a blocking set up to $\dot{\approx}$ under $\mathtt{ack}$.

Supposing $LP \in \mathcal{M}^{M'}_{\Psi,\Phi^-_{aux},\mathtt{ack}:\mathtt{\hat{}^{rw}}T}$ where LP is structural congruent to the process below.

$$\begin{aligned} LP \ \equiv\ & \mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{}^{rw}Site}, \Theta\ \mathbf{in}\ \textstyle\prod_i @_{b_i} c_i\mathtt{!}v_i \\ & \mid @_a([\![R]\!]_a \mid (\mathtt{ack?}p \rightarrow Q) \mid \mathtt{Deliverer}) \end{aligned}$$

and δ is a valid relocator for $((\Psi, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T), M')$. Assume further that

$$\Psi, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T \Vdash LP \xrightarrow[\Delta]{\beta} LQ$$

with β not being an input label with $\mathtt{ack} \in \mathsf{fv}(\beta)$, we shall check all the possibilities that this transition may occur (omitting trivial and similar cases).

Case $R = P_1 \mid P_2$ and P_1 is a `create` or `migrate to`: No transition can be caused by P_1 since there exists no free output on `currentloc`.

Case $R = P_1 \mid P_2$ and $P_1 = c\mathbf{?}p \rightarrow P_1'$ with $c \in \mathsf{dom}(\Psi, \Phi^-_{aux})$: In this case, we have:

- By the definition of $[\![]\!]$, we have $[\![P_1]\!]_a \stackrel{\text{def}}{=} c\mathbf{?}p \rightarrow [\![P_1']\!]_a$.
- We decompose LP as $\mathcal{E}[LQ]$ (and clearly $LP \equiv \mathcal{E}[LQ]$) where

$$\begin{array}{rcl}\mathcal{E}[\cdot] & = & \mathbf{new}\ \mathtt{currentloc} : \mathtt{\hat{}^{rw}Site}, \Theta\ \mathbf{in}\ (\prod_i @_{b_i} c_i\mathbf{!}v_i \\ & & \mid [\cdot] \mid @_a([\![P_2]\!]_a \,|\mathtt{ack}\mathbf{?}p \rightarrow Q|\mathtt{Deliverer})) \\ LQ & = & @_a\,[\![P_1]\!]_a\end{array}$$

- By (LTS-L-IN), (LTS-PRL) and (LTS-NEW), we have

$$\Psi\delta, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T \Vdash LP \xrightarrow[\Delta]{@_a c\mathbf{?}v} LP'$$

  where $LP' = \mathcal{E}[LQ']$ with

$$LQ' \quad \equiv \quad @_a \mathsf{match}(p, v)\, \left[\!\left[P_1'\right]\!\right]_a \ = @_a \left[\!\left[\mathsf{match}(p, v) P_1'\right]\!\right]_a$$

  where $(\Psi\delta, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T) \vdash c \in \mathtt{\hat{}^{r}}S$, $(\Psi\delta, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T, \Delta) \vdash v \in S$ and Δ is an extensible context with $\mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v)$ and $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Theta, \mathtt{currentloc}) = \emptyset$.

- If $\mathtt{ack} \notin \mathsf{fv}(v)$ then $LP' \in \mathcal{M}^{M' \cup \mathsf{mov}(\Delta)}_{\Psi\delta, \Phi^-_{aux}, \mathtt{ack}:\mathtt{\hat{}^{rw}}T, \Delta}$, since $\Psi\delta, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T, \Delta \vdash LP'$ (by Theorem 4.5.1 (Subj)) and the following holds:
  - [Transloc] $M' \cup \mathsf{mov}(\Xi) \subseteq \mathsf{agents}(\Psi\delta, \Xi)/\{a\}$, since $\vdash \Psi\delta, \Phi^-_{aux}, \mathtt{ack} : \mathtt{\hat{}^{rw}}T, \Delta$.
  - [FreeV] $\mathtt{ack}, \mathtt{currentloc} \notin \mathsf{fv}(\mathsf{match}(p, v) P_1' | Q,\ R) \cup \mathsf{fv}(\prod_i @_{b_i} c_i\mathbf{!}v_i)$.

Thus true for this case.

The rest of the analysis is similar to that in the proof of Lemma 6.4.4; we omit the details. ■

Lemma C.1.3 (Blocked by daemon lock (6.4.3))
Given that Γ is a closed located type context, $\mathtt{lock} \notin \mathsf{fv}(P)$ and $M \subseteq \mathsf{agents}(\Gamma)/\{a\}$, the

following processes (each well-typed w.r.t. Γ, Ω_D)

$$
\begin{array}{l}
\textbf{new}\ \Omega_{aux}\ \textbf{in} \\
\quad @_D(Daemon | \prod_i \mathsf{mesgReq}\{T_i\}\ [a\ c_i\ v_i]) \\
\quad |\ @_a([\![P]\!]_a\ |\mathtt{currentloc!}s|\mathtt{Deliverer})
\end{array}
$$

$$
\begin{array}{l}
\textbf{new}\ \Omega_{aux},\ \mathtt{rack} : \mathtt{\char`^rw}[],\ \mathtt{pack} : \mathtt{\char`^rw}[],\ b : \mathtt{Agent}^Z@s\ \textbf{in} \\
\quad @_D(Daemon|\mathsf{regReq}[b\ s\ \mathtt{rack}]| \prod_i \mathsf{mesgReq}\{T_i\}\ [a\ c_i\ v_i]) \\
\quad |\ @_a(\mathsf{regBlockP}(\mathtt{pack}\ Q)|\ [\![P]\!]_a\ |\mathtt{Deliverer})\ |\ @_b\mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ R)
\end{array}
$$

$$
\begin{array}{l}
\textbf{new}\ \Omega_{aux},\ \mathtt{mack} : \mathtt{\char`^rw}[\mathtt{\char`^w}[\mathtt{Site}\ \mathtt{\char`^w}[]]]\ \textbf{in} \\
\quad @_D(Daemon|\mathsf{migReq}[a\ \mathtt{mack}]| \prod_i \mathsf{mesgReq}\{T_i\}\ [a\ c_i\ v_i]) \\
\quad |\ @_a(\mathsf{migBlock}(\mathtt{mack}\ Q)|\ [\![P]\!]_a\ |\mathtt{Deliverer})
\end{array}
$$

are temporary immobile under `lock` w.r.t. $((\Gamma, \Omega_D), M)$.

Proof: Construct a translocating set $\mathcal{M}$ such that $\mathcal{M}^{M'}_{\Psi,\Omega_D}$ is a union of sets of processes of the following forms.

$$
\begin{array}{l}
\textbf{new}\ \Omega_{aux}, \Delta\ \textbf{in}\ (\prod_j @_{b_j} c_j\,\mathtt{!}\,v_j \\
\quad |\ @_D(Daemon\ |\ \prod_i \mathsf{mesgReq}\{T_i\}\ [a_i\ x_i\ v_i']) \\
\quad |\ @_a([\![P']\!]_a\ |\ \mathtt{currentloc!}s\ |\ \mathtt{Deliverer})) \\
\textbf{new}\ \Omega_{aux}, \Delta, \mathtt{pack}, \mathtt{rack} : \mathtt{\char`^rw}[]\ \textbf{in}\ (\prod_j @_{b_j} c_j\,\mathtt{!}\,v_j \\
\quad |\ @_D(Daemon| \prod_i \mathsf{mesgReq}\{T_i\}\ [a_i\ x_i\ v_i']) \\
\quad |\ @_a([\![P']\!]_a\ |\ \mathtt{Deliverer}) \\
\quad |\ @_a(\textbf{create}^Z\ b = \langle D@SD\rangle\mathtt{register!}[b\ s\ \mathtt{rack}]\ |\ \mathsf{regBlockC}(\mathtt{pack}\ \mathtt{rack}\ Q') \\
\qquad \textbf{in}\ \mathsf{regBlockP}(\mathtt{pack}\ R'))) \\
\textbf{new}\ \Omega_{aux}, \Delta, \mathtt{pack}, \mathtt{rack} : \mathtt{\char`^rw}[], b : \mathtt{Agent}^Z@s\ \textbf{in}\ (\prod_j @_{b_j} c_j\,\mathtt{!}\,v_j \\
\quad |\ @_D(Daemon\ |\ \mathsf{regReq}[b\ s\ \mathtt{pack}]\ |\ \prod_i \mathsf{mesgReq}\{T_i\}\ [a_i\ x_i\ v_i']) \\
\quad |\ @_a(\mathsf{regBlockP}(\mathtt{pack}\ R')\ |\ [\![P']\!]_a\ |\ \mathtt{Deliverer})\ |\ @_b\mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ Q')) \\
\textbf{new}\ \Omega_{aux}, \Delta, \mathtt{mack} : \mathtt{\char`^rw}[\mathtt{\char`^w}[\mathtt{Site}\ \mathtt{\char`^w}[]]]\ \textbf{in}\ (\prod_j @_{b_j} c_j\,\mathtt{!}\,v_j \\
\quad |\ @_D(Daemon\ |\ \mathsf{migReq}[a\ \mathtt{mack}]\ |\ \prod_i \mathsf{mesgReq}\{T_i\}\ [a_i\ x_i\ v_i']) \\
\quad |\ @_a(\mathsf{migBlock}(\mathtt{mack}\ Q')\ |\ [\![P']\!]_a\ |\ \mathtt{Deliverer}))
\end{array}
$$

well-typed wrt Ψ, Ω_D, satisfying

$$M' \subseteq \mathsf{agents}(\Psi)/\{a\} \qquad ([\text{Transloc}])$$

$$
\begin{array}{l}
(\{\mathtt{lock}, \mathtt{pack}, \mathtt{rack}, \mathtt{mack}\} \cup \mathsf{dom}(\Omega_{aux})) \\
\quad \cap(\mathsf{fv}(P', Q', R', \prod_j @_{b_j} c_j\,\mathtt{!}\,v_j, \prod_i \mathsf{mesgReq}\{T_i\}\ [a_i\ x_i\ v_i'])) = \emptyset
\end{array}
\qquad ([\text{FreeV}])
$$

We shall prove that $\mathcal{M}$ is a blocking set up to $\dot{\approx}$ under `lock`,

Supposing $LP \in \mathcal{M}^{M'}_{\Psi,\Omega_D}$ and δ is a valid relocator for $((\Psi,\Omega_D),M')$. Assume further that $\Psi,\Omega_D \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ with β not being an input label with $\mathtt{lock} \in \mathsf{fv}(\beta)$, we shall check all the possibilities that this transition may occur (omitting trivial and similar cases).

Case Initialising creation: Supposing LP is structurally congruent to the process below (which belongs to the first subset).

$$\begin{aligned}
&\mathbf{new}\ \Omega_{aux},\Delta\ \mathbf{in}\ (\textstyle\prod_j @_{b_j}c_j\,\mathtt{!}\,v_j\\
&\quad \mid @_D(Daemon \mid \textstyle\prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i'])\\
&\quad \mid @_a([\![P_1]\!]_a \mid [\![P']\!]_a \mid \mathtt{currentloc!}s \mid \mathtt{Deliverer}))
\end{aligned}$$

with $P_1 = \mathbf{create}^Z\ b = Q'\ \mathbf{in}\ R'$. In this case, we have:

- By (Lts-L-Comm), (Lts-Prl) and (Lts-New), $\Psi\delta,\Omega_D \Vdash LP \xrightarrow{\tau} LQ$ where LQ is structurally congruent to the process below.

$$\begin{aligned}
&\mathbf{new}\ \Omega_{aux},\Delta,\mathtt{pack},\mathtt{rack}:\mathtt{\hat{}rw}[]\ \mathbf{in}\ (\textstyle\prod_j @_{b_j}c_j\,\mathtt{!}\,v_j\\
&\quad \mid @_D(Daemon \mid \textstyle\prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i'])\\
&\quad \mid @_a([\![P']\!]_a \mid \mathtt{Deliverer})\\
&\quad \mid @_a(\mathbf{create}^Z\ b = \langle D@SD\rangle\mathtt{register!}[b\ s\ \mathtt{rack}] \mid \mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ Q')\\
&\qquad \mathbf{in}\ \mathsf{regBlockP}(\mathtt{pack}\ R')))
\end{aligned}$$

- Clearly, $LQ \in \mathcal{M}^{M'}_{\Psi\delta,\Omega_D}$ since $\Psi\delta,\Omega_D \vdash LQ$ (by Theorem 4.5.1 (Subj)) and the following holds:
  - [Transloc] $M' \subseteq \mathsf{agents}(\Psi\delta)/\{a\}$.
  - [FreeV] $(\mathtt{lock}\cup\mathsf{dom}(\Omega_{aux}))\cap\mathsf{fv}(P_1 \mid P') = \emptyset$ implies $(\{\mathtt{lock},\mathtt{pack},\mathtt{rack}\}\cup\mathsf{dom}(\Omega_{aux}))\cap(\mathsf{fv}(P',Q',R') \cup \mathsf{fv}(\prod_j @_{b_j}c_j\,\mathtt{!}\,v_j, \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i']) = \emptyset$, since `pack`, `rack` are fresh channels.

Thus true for this case.

Case Initialising migration: Supposing LP is structurally congruent to the process below (which belongs to the first subset).

$$\begin{aligned}
&\mathbf{new}\ \Omega_{aux},\Delta\ \mathbf{in}\ (\textstyle\prod_j @_{b_j}c_j\,\mathtt{!}\,v_j\\
&\quad \mid @_D(Daemon \mid \textstyle\prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i'])\\
&\quad \mid @_a([\![P_1]\!]_a \mid [\![P']\!]_a \mid \mathtt{currentloc!}s \mid \mathtt{Deliverer}))
\end{aligned}$$

with $P_1 = \mathbf{migrate\ to}\ s \to Q'$. In this case, we have:

- By (LTS-L-COMM), (LTS-PRL) and (LTS-NEW), $\Psi\delta, \Omega_D \Vdash LP \xrightarrow{\tau} LQ$ where LQ is structural congruent to the process below.

$$\begin{array}{l}\textbf{new } \Omega_{aux}, \Delta, \texttt{mack} : \texttt{^rw}[\texttt{^w}[\texttt{Site ^w}[]]] \textbf{ in } (\prod_j @_{b_j} c_j \,!\, v_j \\ \quad \mid @_D(\mathit{Daemon} \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i']) \\ \quad \mid @_a(\llbracket P' \rrbracket_a \mid \texttt{Deliverer}) \\ \quad \mid @_a(\langle D@SD\rangle \texttt{migrating!}\,[a\ \texttt{mack}] \mid \mathsf{migBlock}(\texttt{mack}\ Q')))\end{array}$$

- By Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), we have

$$@_a\langle D@SD\rangle \texttt{migrating!}\,[a\ \texttt{mack}] \quad \succeq_\Phi \quad @_D\texttt{migrating!}\,[a\ \texttt{mack}]$$

where $\Phi = \Psi, \Omega_D, \Omega_{aux}, \Delta, \texttt{mack} : \texttt{^rw}[\texttt{^w}[\texttt{Site ^w}[]]]$, since D is a static agent (ie. $D \notin \mathsf{mov}(\Phi)$).

- By Theorem 5.4.4 (Transloc Bisim Cong), we have $LQ \dot{\succeq}^{M'}_{\Psi\delta,\Omega_D} LQ_1$, where LQ_1 is structural congruent to the process below.

$$\begin{array}{l}\textbf{new } \Omega_{aux}, \Delta, \texttt{mack} : \texttt{^rw}[\texttt{^w}[\texttt{Site ^w}[]]] \textbf{ in } (\prod_j @_{b_j} c_j \,!\, v_j \\ \quad \mid @_D(\mathit{Daemon} \mid \texttt{migrating!}\,[a\ \texttt{mack}] \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i']) \\ \quad \mid @_a(\llbracket P' \rrbracket_a \mid \texttt{Deliverer}) \mid @_a\mathsf{migBlock}(\texttt{mack}\ Q'))\end{array}$$

- By [FreeV], applying Lemma 6.2.3, we have $LQ_1 \dot{\succeq}^{M'}_{\Psi\delta,\Omega_D} LQ_2$, where LQ_2 is structural congruent to the process below.

$$\begin{array}{l}\textbf{new } \Omega_{aux}, \Delta, \texttt{mack} : \texttt{^rw}[\texttt{^w}[\texttt{Site ^w}[]]] \textbf{ in } (\prod_j @_{b_j} c_j \,!\, v_j \\ \quad \mid @_D(\mathit{Daemon} \mid \mathsf{migReq}(a\ \texttt{mack}) \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i']) \\ \quad \mid @_a(\llbracket P' \rrbracket_a \mid \texttt{Deliverer}) \mid @_a\mathsf{migBlock}(\texttt{mack}\ Q'))\end{array}$$

- Clearly, $LQ \in \mathcal{M}^{M'}_{\Psi\delta,\Omega_D}$ since $\Psi\delta, \Omega_D \vdash LQ$ (by Theorem 4.5.1 (Subj)) and the following holds
  - [Transloc] $M' \subseteq \mathsf{agents}(\Psi\delta)/\{a\}$.
  - [FreeV] By Theorem 4.5.1 (Subj), $(\texttt{lock} \cup \mathsf{dom}(\Omega_{aux})) \cap \mathsf{fv}(P_1 \mid P') = \emptyset$ implies $(\{\texttt{lock}, \texttt{mack}\} \cup \mathsf{dom}(\Omega_{aux})) \cap$ $(\mathsf{fv}(P', Q') \cup \mathsf{fv}(\prod_i @_{b_i} c_i \,!\, v_i, \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v_i'])) = \emptyset$, since $\texttt{mack}$ is fresh.

Thus true for this case.

Case Sending registration: Supposing LP is structurally congruent to the process below (which belongs to the second subset).

$$\begin{array}{l}\textbf{new}\ \Omega_{aux}, \Delta, \texttt{pack}, \texttt{rack} : \texttt{\^{}rw}[]\ \textbf{in}\ (\prod_j @_{b_j} c_j\,!\,v_j \\ \quad \mid @_D(\mathit{Daemon} \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v'_i]) \\ \quad \mid @_a([\![P']\!]_a \mid \texttt{Deliverer}) \\ \quad \mid @_a(\textbf{create}^Z\ b = \langle D@SD\rangle\texttt{register!}[b\ s\ \texttt{rack}] \mid \mathsf{regBlockC}(\texttt{rack}\ \texttt{pack}\ Q') \\ \qquad \textbf{in}\ \mathsf{regBlockP}(\texttt{pack}\ R')))\end{array}$$

In this case, we have:

- By (LTS-L-CREATE), (LTS-PRL) and (LTS-NEW), $\Psi\delta, \Omega_D \Vdash LP \xrightarrow{\tau} LQ$ where LQ is structurally congruent to the process below.

$$\begin{array}{l}\textbf{new}\ \Omega_{aux}, \Delta, b : \texttt{Agent}^Z@s, \texttt{pack}, \texttt{rack} : \texttt{\^{}rw}[]\ \textbf{in}\ (\prod_j @_{b_j} c_j\,!\,v_j \\ \quad \mid @_D(\mathit{Daemon} \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v'_i]) \\ \quad \mid @_a(\mathsf{regBlockP}(\texttt{pack}\ R') \mid [\![P']\!]_a \mid \texttt{Deliverer}) \\ \quad \mid @_b(\langle D@SD\rangle\texttt{register!}[b\ s\ \texttt{rack}] \mid \mathsf{regBlockC}(\texttt{rack}\ \texttt{pack}\ Q')))\end{array}$$

- By Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), we have:

$$@_a\langle D@SD\rangle\texttt{register!}[b\ s\ \texttt{rack}] \quad \succeq_\Phi \quad @_D\texttt{register!}[b\ s\ \texttt{rack}]$$

  where $\Phi = \Psi, \Omega_D, \Omega_{aux}, \Delta, b : \texttt{Agent}^Z@s, \texttt{pack}, \texttt{rack} : \texttt{\^{}rw}[]$, since D is a static agent (ie. $D \notin \mathsf{mov}(\Phi)$).

- By Theorem 5.4.4 (Transloc Bisim Cong), we have $LQ \dot{\succeq}^{M'}_{\Psi\delta,\Omega_D} LQ_1$, where LQ_1 is structurally congruent to the process below.

$$\begin{array}{l}\textbf{new}\ \Omega_{aux}, \Delta, b : \texttt{Agent}^Z@s, \texttt{pack}, \texttt{rack} : \texttt{\^{}rw}[]\ \textbf{in}\ (\prod_j @_{b_j} c_j\,!\,v_j \\ \quad \mid @_D(\mathit{Daemon} \mid \texttt{register!}[b\ s\ \texttt{rack}] \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v'_i]) \\ \quad \mid @_a(\mathsf{regBlockP}(\texttt{pack}\ R') \mid [\![P']\!]_a \mid \texttt{Deliverer}) \\ \quad \mid @_b\mathsf{regBlockC}(\texttt{rack}\ \texttt{pack}\ Q'))\end{array}$$

- By [FreeV], applying Lemma 6.2.3, we have $LQ_1 \dot{\succeq}^{M'}_{\Psi\delta,\Omega_D} LQ_2$, where LQ_2 is structurally congruent to the process below.

$$\begin{array}{l}\textbf{new}\ \Omega_{aux}, \Delta, b : \texttt{Agent}^Z@s, \texttt{pack}, \texttt{rack} : \texttt{\^{}rw}[]\ \textbf{in}\ (\prod_j @_{b_j} c_j\,!\,v_j \\ \quad \mid @_D(\mathit{Daemon} \mid \mathsf{regReq}(b\ s\ \texttt{rack}) \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v'_i]) \\ \quad \mid @_a(\mathsf{regBlockP}(\texttt{pack}\ R') \mid [\![P']\!]_a \mid \texttt{Deliverer}) \\ \quad \mid @_b\mathsf{regBlockC}(\texttt{rack}\ \texttt{pack}\ Q'))\end{array}$$

- Clearly, $LQ \in \mathcal{M}^{M'}_{\Psi\delta,\Omega_D}$ since $\Psi\delta, \Omega_D \vdash LQ$ (by Theorem 4.5.1 (Subj)) and the following holds:
  - [Transloc] $M' \subseteq \mathsf{agents}(\Psi\delta)/\{a\}$.
  - [FreeV] The condition remains unchanged: $(\{\mathtt{lock}, \mathtt{pack}, \mathtt{rack}\} \cup \mathsf{dom}(\Omega_{aux})) \cap (\mathsf{fv}(P', Q', R') \cup \mathsf{fv}(\prod_j @_{b_j} c_j \,!\, v_j, \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v'_i]) = \emptyset$.

Thus true for this case.

Case Message forwarding: Supposing LP can be decomposed as $\mathcal{E}[LQ]$ (and clearly $LP \equiv \mathcal{E}[LQ]$), where

$$\begin{array}{rcl} \mathcal{E}[\cdot] & = & \mathbf{new}\ \Omega_{aux}, \Delta\ \mathbf{in}\ (\prod_{j\neq 1} @_{b_j} c_j \,!\, v_j \\ & & \quad \mid @_D(Daemon \mid \prod_i \mathsf{mesgReq}\{T_i\}\,[a_i\ x_i\ v'_i]) \\ & & \quad \mid [\cdot] \mid @_a(\llbracket P' \rrbracket_a \mid \mathtt{currentloc!}\,s \mid \mathtt{Deliverer})) \\ LQ & = & @_a \llbracket \langle b@?\rangle c \,!\, v \rrbracket_a \end{array}$$

Note that here we pick LP from the first subset, but $\mathcal{E}[\cdot]$ can be redefined so that LP belongs to other subsets. In this case, we have:

- By the definition of $\llbracket\cdot\rrbracket_a$, we have: $\llbracket Q \rrbracket_a = \langle D@SD\rangle \mathtt{message!}\ \{T\}\,[b\ c\ v]$, where T is the type of v in the context.
- By Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), we have:

$$@_a\langle D@SD\rangle \mathtt{message!}\ \{T\}\,[b\ c\ v] \quad \succeq_{\mathcal{E}[\Psi\delta,\Omega_D]} \quad @_D \mathtt{message!}\ \{T\}\,[b\ c\ v]$$

  since D is a static agent (ie. $D \notin \mathsf{mov}(\mathcal{E}[\Psi\delta, \Omega_D])$).

- Since $D \notin \mathsf{mayMove}(\mathcal{E}[\cdot])$, by Theorem 5.4.4 (Transloc Bisim Cong), we have:

$$\mathcal{E}[LQ] \dot{\succeq}^{M'}_{\Psi\delta,\Omega_D} \mathcal{E}[@_D \mathtt{message!}\ \{T\}\,[b\ c\ v]]$$

- By [FreeV], applying Lemma 6.2.3, we have

$$\mathcal{E}[@_D \mathtt{message!}\ \{T\}\,[b\ c\ v]] \dot{\succeq}^{M'}_{\Psi\delta,\Omega_D} \mathcal{E}[@_D \mathsf{mesgReq}(\{T\}\,[b\ c\ v])]$$

- Clearly, $\mathcal{E}[@_D \mathsf{mesgReq}(\{T\}\,[b\ c\ v])] \in \mathcal{M}^{M'}_{\Psi\delta,\Omega_D}$ since $\Psi\delta, \Omega_D \vdash \mathcal{E}[@_D \mathsf{mesgReq}(\{T\}\,[b\ c\ v])]$ and the following holds
  - [Transloc] $M' \subseteq \mathsf{agents}(\Psi\delta)/\{a\}$.

– [FreeV] $(\mathtt{lock} \cup \mathsf{dom}(\Omega_{aux})) \cap \mathsf{fv}(P' \mid \langle b@?\rangle c\,\mathtt{!}\,v) = \emptyset$ implies $(\mathtt{lock} \cup \mathsf{dom}(\Omega_{aux})) \cap \mathsf{fv}(\mathsf{mesgReq}\{T\}\,[b\ c\ v]) = \emptyset$.

Thus true for this case.

Case Internal communication: We may observe that, by [FreeV], `lock`, `pack`, `rack` and `mack` are never involved in any internal communication. The details of this case is similar to that analysed in the proof of Lemma 6.4.4 (omitted).

Case Input via `deliver`: Similar to that analysed in the proof of Lemma 6.4.4. Note, however, that the acknowledgement $@_D\mathtt{dack!}[]$ will not react with *Daemon*, but be added to the queue of outgoing messages $\prod_j @_{b_j} c_j\,\mathtt{!}\,v_j$.

Case $P' = P_1 \mid P_2$ and $P_1 =$ `iflocal` $\langle b\rangle c\,\mathtt{!}\,v$ `in` P_1' `else` P_1'' with $\Psi\delta, \Phi_{aux} \vdash a@s \wedge b@s$ and others: We omit these cases, since they are similar to those analysed in the proof of Lemma 6.4.4.

■

Appendix D

Correctness Proof

D.1 General Properties

Lemma D.1.1 (Sys str cong preserves well-formedness)
Given that $Sys \equiv_{\Phi} Sys'$, $\Phi \vdash Sys\ ok$ if and only if $\Phi \vdash Sys'\ ok$.

Proof: We shall only demonstrate that if the structural congruence is derived using (SYS-STR-LOCAL-DEC) then such congruence preserves well-formedness.

Supposing $\Phi \vdash \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})\ ok$ and $\boldsymbol{A}(a) = [(\mathbf{new}\ \Theta\ \mathbf{in}\ P)\ \boldsymbol{E}]$ with $\boldsymbol{D} = [map\ \mathsf{mesgQ}]$. We may derive the following. $\Phi, \Delta \vdash map\ ok$, $\Phi, \Delta \vdash \mathsf{mesgQ}\ ok$, $\Phi, \Delta \vdash \boldsymbol{A}\ ok$, $\vdash \Phi\ ok$ and $\mathsf{dom}(\boldsymbol{A}) = \mathsf{dom}(map)$, by (SYS-T-EPROG). $\Phi, \Delta \vdash_a \mathbf{new}\ \Theta\ \mathbf{in}\ P$, $\Phi, \Delta \vdash_a \boldsymbol{E}\ ok$ etc, by (SYS-T-ASTATE). $\mathsf{dom}(\Theta) \cap \mathsf{dom}(\Phi, \Delta) = \emptyset$ implies $\Phi, \Delta, \Theta \vdash_a P$, by (LOCALNEW); moreover, Θ only declare channel names. $\Phi, \Delta, \Theta \vdash map\ ok$, $\Phi, \Delta, \Theta \vdash \mathsf{mesgQ}\ ok$, and $\Phi, \Delta, \Theta \vdash \boldsymbol{A} \oplus a \mapsto [P\ \boldsymbol{E}]$, by Lemma 3.7.2 (Type SW). $\Phi \vdash \mathsf{eProg}(\Delta, \Theta; \boldsymbol{D}; \boldsymbol{A} \oplus a \mapsto [P\ \boldsymbol{E}])\ ok$, by (SYS-T-EPROG). Thus true for the left-to-right direction. The righ-to-left direction can be obtained in a similar manner. ■

Lemma D.1.2 (Flattening preserves well-formedness)
Given a valid system context Φ, if $Sys \equiv_{\Phi} Sys'$ then $\mathcal{F}[\![Sys]\!] \equiv \mathcal{F}[\![Sys']\!]$.

Lemma D.1.3 (Potentially movable agents in $\mathcal{F}$)
Given a valid system context Φ, if $\Phi \vdash \boldsymbol{A}\ ok$ then $\mathsf{mayMove}(\mathcal{F}[\![\boldsymbol{A}]\!]) \subseteq \mathsf{dom}(\boldsymbol{A})$.
Moreover, $\mathsf{mayMove}(@_D(Daemon \mid \mathsf{mesgQ})) = \emptyset$, $\mathsf{mayMove}(\mathsf{mapS}_{\bullet}(map)) = \emptyset$ and $\mathsf{mayMove}(\mathsf{mapS}_{(a_e,\ \boldsymbol{A}(a_e))}(map)) = \{a_e\}$

Lemma D.1.4 (Accuracy of location map)
For any $Sys = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$, with $\Phi \vdash Sys\ ok$, $\mathsf{lookupL}(a; map) = s$ implies $\Phi, \Delta \vdash a@s$.

Lemma D.1.5 (Subject reduction for IL (7.2.1))
Given a valid system context Φ, if $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ then $\Phi, \Xi \vdash Sys'\ ok$.

Proof: Assuming $\Phi \vdash Sys\ ok$ with $Sys = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$ and $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$, we shall prove that $\Phi, \Xi \vdash Sys'\ ok$ by an induction on the transition derivation, demonstrating some interesting cases.

Case (Sys-Loc-Replic): In this case, the following conditions are satisfied:

- $a, c \in \mathsf{dom}(\Phi)$, $\boldsymbol{A}(a) = [(*c?p \to Q) \mid R\ \boldsymbol{E}]$ and
- $\Phi \vdash c \in \hat{}^{\mathbf{r}}T$, $(\Phi, \Xi) \vdash v \in T$, $\mathsf{dom}(\Xi) \subseteq \mathsf{fv}(v)$, Ξ is extensible and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta) = \emptyset$.

This implies, by (Sys-Loc-Replic), $\Phi \Vdash Sys \xrightarrow[\Xi]{@_a c?v} Sys'$ where

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a \mapsto [\mathsf{match}(p, v)Q \mid *c?p \to Q \mid R\ \boldsymbol{E}])$$

The following can be deduced. By (Sys-T-EProg), $\Phi, \Delta \vdash map\ ok$, $\Phi, \Delta \vdash \boldsymbol{A}\ ok$, $\Phi, \Delta \vdash \mathsf{mesgQ}\ ok$, $\vdash \Phi\ ok$ and $\mathsf{dom}(\boldsymbol{A}) = \mathsf{dom}(map)$. By (Sys-T-AState), $\Phi, \Delta \vdash_a (*c?v \to Q) \mid R$ and $\Phi, \Delta \vdash_a \boldsymbol{E}\ ok$. By (Par), $\Phi, \Delta \vdash_a *c?p \to Q$ and $\Phi, \Delta \vdash_a R$. By (Replic), $\Phi, \Delta \vdash p \in T \triangleright \Theta$ and $\Phi, \Delta, \Theta \vdash_a Q$. $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta) = \emptyset$ and $\vdash \Phi, \Xi$ implies $\vdash \Phi, \Delta, \Xi$, by (L-C-Var). By Lemma B.1.9 (Type Match), $\Phi, \Delta, \Xi \vdash_a \mathsf{match}(p, v)Q$. Given that $\vdash \Phi\ ok$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, we have, by Lemma 7.2.2, $\vdash \Phi, \Xi\ ok$. By (Sys-T-AState), $\Phi, \Xi, \Delta \vdash \boldsymbol{A} \oplus a \mapsto [\mathsf{match}(p, v)Q \mid *c?p \to Q \mid R\ \boldsymbol{E}]\ ok$. By (Sys-T-EProg), $\Phi \vdash Sys'\ ok$. Thus true for this case.

Case (Sys-Loc-Out): In this case, we pick an injective substitution $\sigma : \mathsf{dom}(\Delta) \to \mathcal{X}/\mathsf{dom}(\Phi, \Phi_{aux})$ in order to prevent the names in Δ (which might be extruded) from clashing with names defined in Φ_{aux}. We have $Sys \stackrel{\alpha}{=} \mathsf{eProg}(\Delta'; [map'\ \mathsf{mesgQ}']; \boldsymbol{A}')$, where $\Delta' = \sigma\Delta$, $map' = \sigma map$, $\mathsf{mesgQ}' = \sigma\mathsf{mesgQ}$ and $\boldsymbol{A}' = \sigma\boldsymbol{A}$. Assume further that the following conditions are satisfies:

- $a, c \in \mathsf{dom}(\Phi)$, $\boldsymbol{A}'(a) = [c!v \mid R\ \boldsymbol{E}]$ and
- $\Delta' \equiv \Delta_1, \Delta_2$ and $\mathsf{dom}(\Delta') \cap \mathsf{fv}(v) = \mathsf{dom}(\Delta_1)$.

This implies, by (Sys-Loc-Replic) (and (Sys-Equiv)), $\Phi \Vdash Sys \xrightarrow[\Delta_1]{@_a c!v} Sys'$ where

$$Sys' \quad = \quad \mathsf{eProg}(\Delta_2; [map'\ \mathsf{mesgQ}']; \boldsymbol{A}' \oplus a \mapsto [R\ \boldsymbol{E}])$$

The following can be deduced. By (Sys-T-EProg), $\Phi, \Delta' \vdash map'\ ok$, $\Phi, \Delta' \vdash \boldsymbol{A}'\ ok$, $\Phi, \Delta' \vdash \mathsf{mesgQ}'\ ok$, $\vdash \Phi\ \ ok$ and $\mathsf{dom}(\boldsymbol{A}') = \mathsf{dom}(map')$. By (Sys-T-AState), $\Phi, \Delta' \vdash_a c\mathtt{!}v \mid R$ and $\Phi, \Delta' \vdash_a \boldsymbol{E}\ ok$. By (Par), $\Phi, \Delta' \vdash_a c\mathtt{!}v$ and $\Phi, \Delta' \vdash_a R$. By (Out), $\Phi, \Delta' \vdash c \in \mathtt{\hat{}^{w}}T$ and $\Phi, \Delta' \vdash v \in T$. Given that $\vdash \Phi\ ok$, $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ implies, by Lemma 7.2.2, $\vdash \Phi, \Delta_1\ ok$. By (Sys-T-AState), $\Phi, \Delta' \vdash \boldsymbol{A}' \oplus a \mapsto [R\ \boldsymbol{E}]\ ok$. By (Sys-T-EProg), $\Phi, \Delta_1 \vdash Sys'\ ok$. Thus true for this case.

Case (Sys-Req-Reg): In this case, the following conditions are satisfies:

- $\boldsymbol{A}(a) = [(\mathtt{\textbf{create}}^Z\ b = P\ \mathtt{\textbf{in}}\ Q) \mid R\ \mathsf{FreeA}(s)]$, and $b \notin \mathsf{dom}(\Phi, \Delta)$.

This implies, by (Sys-Req-Reg), $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ where

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a \mapsto [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)])$$

The following can be deduced. By (Sys-T-EProg), $\Phi, \Delta \vdash map\ ok$, $\Phi, \Delta \vdash \boldsymbol{A}\ ok$, $\Phi, \Delta \vdash \mathsf{mesgQ}\ ok$, $\vdash \Phi\ ok$ and $\mathsf{dom}(\boldsymbol{A}) = \mathsf{dom}(map)$. By (Sys-T-AState), $\Phi, \Delta \vdash_a (\mathtt{\textbf{create}}^Z\ b = P\ \mathtt{\textbf{in}}\ Q) \mid R$ and $\Phi, \Delta \vdash_a \mathsf{FreeA}(s)\ ok$. By (Sys-T-FreeA), $\Phi, \Delta \vdash a@s$. By (Par), $\Phi, \Delta \vdash_a \mathtt{\textbf{create}}^Z\ b = P\ \mathtt{\textbf{in}}\ Q$ and $\Phi, \Delta \vdash_a R$. Since $b \notin \mathsf{dom}(\Phi, \Delta)$, we have $\vdash \Phi, \Delta, b : \mathtt{Agent}^Z@s$, by (L-C-Var). By (Create), $\Phi, \Delta, b : \mathtt{Agent}^Z@s \vdash_a P, Q$ and $a \neq b$. By (Sys-T-RegA), $\Phi, \Delta \vdash_a \mathsf{RegA}(b\ Z\ s\ P\ Q)\ ok$. By (Sys-T-AState), $\Phi, \Delta \vdash \boldsymbol{A} \oplus a \mapsto [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)]\ ok$. By (Sys-T-EProg), $\Phi \vdash Sys'\ ok$. Thus true for this case.

Case (Sys-Proc-Reg): In this case, the following conditions are satisfies:

- $\boldsymbol{A}(a) = [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)]$, Sys is idle, and $b \notin \mathsf{dom}(\Phi, \Delta)$.

This implies, by (Sys-Proc-Reg), $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ where Sys' denotes the following process.

$$\mathsf{eProg}(\Delta, b : \mathtt{Agent}^Z@s; [[b\ s]\mathtt{::}map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a \mapsto [Q|R\ \mathsf{FreeA}(s)] \oplus b \mapsto [P\ \mathsf{FreeA}(s)])$$

The following can be deduced. By (Sys-T-EProg), $\Phi, \Delta \vdash map\ ok$, $\Phi, \Delta \vdash \boldsymbol{A}\ ok$, $\Phi, \Delta \vdash \mathsf{mesgQ}\ ok$, $\vdash \Phi\ ok$ and $\mathsf{dom}(\boldsymbol{A}) = \mathsf{dom}(map)$. By (Sys-T-AState), $\Phi, \Delta \vdash_a R$ and $\Phi, \Delta \vdash_a \mathsf{RegA}(b\ Z\ s\ P\ Q)\ ok$. Since $b \notin \mathsf{dom}(\Phi, \Delta)$, we have $\vdash \Phi, \Delta, b : \mathtt{Agent}^Z@s$, by (L-C-Var). By (Sys-T-RegA), $\Phi, \Delta \vdash a@s$, $(\Phi, \Delta, b : \mathtt{Agent}^Z@s) \vdash_b P$ and $(\Phi, \Delta, b : \mathtt{Agent}^Z@s) \vdash_a Q$. By Lemma 3.7.2 (Type SW), $(\Phi, \Delta, b : \mathtt{Agent}^Z@s) \vdash_b R$. By (sys-T-FreeA), $\Phi, \Delta, b : \mathtt{Agent}^Z@s \vdash_b \mathsf{FreeA}(s)$ and $\Phi, \Delta, b : \mathtt{Agent}^Z@s \vdash_a \mathsf{FreeA}(s)$. By (Sys-T-AState), $\Phi, \Delta, b : \mathtt{Agent}^Z@s \vdash \boldsymbol{A} \oplus a \mapsto [Q|R\ \mathsf{FreeA}(s)] \oplus b \mapsto [P\ \mathsf{FreeA}(s)]\ ok$. $\Phi, \Delta, b : \mathtt{Agent}^Z@s \vdash \mathsf{mesgQ}\ ok$, by Lemma 3.7.2 (Type SW). $\mathsf{dom}(\boldsymbol{A} \oplus a \mapsto [Q|R\ \mathsf{FreeA}(s)] \oplus b \mapsto [P\ \mathsf{FreeA}(s)]) \subseteq \mathsf{dom}([b\ s]\mathtt{::}map)$. By (Sys-T-Map), there exists a_i, s_i's such that

$$\mathsf{consolidate}(map) = [a_1\ s_1]\mathtt{::} \ldots \mathtt{::}[a_n\ s_n]\mathtt{::\textbf{nil}}$$

with $\{a_1, \ldots, a_n\} = \mathsf{agents}(\Phi, \Delta)$ and, for all j, $\Phi, \Delta \vdash a_j @ s_j$. Since $b \notin \mathsf{dom}(\Phi, \Delta)$, we have:

$$\mathsf{consolidate}([b\ s] \mathbin{::} map) = [b\ s] \mathbin{::} [a_1\ s_1] \mathbin{::} \ldots \mathbin{::} [a_n\ s_n] \mathbin{::} \mathbf{nil}$$

with $\{b, a_1, \ldots, a_n\} = \mathsf{agents}(\Phi, \Delta, b : \mathtt{Agent}^Z @ s)$ and, for all j, $\Phi, \Delta, b : \mathtt{Agent}^Z @ s \vdash a_j @ s_j$, by Lemma 3.7.2 (Type SW). By (Sys-T-Map), $\Phi, \Delta, b : \mathtt{Agent}^Z @ s \vdash [b\ s] \mathbin{::} map\ ok$. By (Sys-T-EProg), $\Phi \vdash Sys'\ ok$. Thus true for this case.

Case (Sys-Comm-Mig): In this case, the following conditions are satisfies:

- $\boldsymbol{A}(a) = [R\ \mathsf{MrdyA}(s\ P)]$ and $a \in \mathsf{dom}(\Delta)$.

This implies, by (Sys-Proc-Reg), $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ where Sys' denotes the following.

$$\mathsf{eProg}(\Delta \oplus a \mapsto s; [[a\ s] \mathbin{::} map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a \mapsto [P|R\ \mathsf{FreeA}(s)])$$

The following can be deduced. By (Sys-T-EProg), $\Phi, \Delta \vdash map\ ok$, $\Phi, \Delta \vdash \boldsymbol{A}\ ok$, $\Phi, \Delta \vdash \mathsf{mesgQ}\ ok$, $\vdash \Phi\ ok$ and $\mathsf{dom}(\boldsymbol{A}) = \mathsf{dom}(map)$. By (Sys-T-AState), $\Phi, \Delta \vdash_a R$ and $\Phi, \Delta \vdash_a \mathsf{MrdyA}(s\ P)\ ok$. By (Sys-T-MtedA), $\Phi, \Delta \vdash s \in \mathtt{Site}$, $\Phi, \Delta \vdash_a P$. By (sys-T-FreeA), $\Phi, \Delta \oplus a \mapsto s \vdash_a \mathsf{FreeA}(s)$. By Lemma B.1.2, $\Phi, \Delta \oplus a \mapsto s \vdash_a R$. By (Sys-T-AState), $\Phi, \Delta \oplus a \mapsto s \vdash \boldsymbol{A} \oplus a \mapsto [P|R\ \mathsf{FreeA}(s)]\ ok$. $\vdash \Phi, \Delta \oplus a \mapsto s$, by Lemma B.1.2. $\Phi, \Delta \oplus a \mapsto s \vdash \mathsf{mesgQ}\ ok$, by Lemma B.1.2. $\mathsf{dom}(\boldsymbol{A} \oplus a \mapsto [P|R\ \mathsf{FreeA}(s)]) \subseteq \mathsf{dom}([a\ s] \mathbin{::} map)$ By (Sys-T-Map), there exists a_i, s_i's such that

$$\begin{aligned} \mathsf{consolidate}(map) &= [a_1\ s_1] \mathbin{::} \ldots \mathbin{::} [a_n\ s_n] \mathbin{::} \mathbf{emptymap} \\ \{a_1, \ldots, a_n\} &= \mathsf{agents}(\Phi, \Delta) \end{aligned}$$

and, for all j, $\Phi, \Delta \vdash a_j @ s_j$. Assuming $a = a_k$, by the definition of $\mathsf{consolidate}(\cdot)$,

$$\mathsf{consolidate}([a\ s] \mathbin{::} map) = [a\ s] \mathbin{::} [a_1\ s_1] \mathbin{::} \ldots \mathbin{::} [a_{k-1}\ s_{k-1}] \mathbin{::} [a_{k+1}\ s_{k+1}] \mathbin{::} \ldots \mathbin{::} [a_n\ s_n] \mathbin{::} \mathbf{emptymap}$$

Moreover, $\{a, a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n\} = \mathsf{agents}(\Phi, \Delta \oplus a \mapsto s)$ and, for all $j \neq k$, $\Phi, \Delta @_a \mathsf{migrate\ to}\ s \vdash a_j @ s_j$, by Lemma B.1.2. By (Sys-T-Map), $\Phi, \Delta \oplus a \mapsto s \vdash [a\ s] \mathbin{::} map\ ok$. By (Sys-T-EProg), $\Phi \vdash Sys'\ ok$. Thus true for this case.

Inductive Case (Sys-Equiv): In this case, we have $Sys_1 \equiv_\Phi Sys_2$, $Sys'_1 \equiv_{\Phi,\Xi} Sys'_2$ and $\Phi \Vdash Sys'_1 \xrightarrow[\Xi]{\beta} Sys'_2$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ implies, by (Sys-Equiv), $\Phi \Vdash Sys_1 \xrightarrow[\Xi]{\beta} Sys_2$. $\Phi \vdash Sys_1\ ok$ implies $\Phi \vdash Sys'_1\ ok$, by Lemma D.1.1. $\Phi, \Xi \vdash Sys'_2\ ok$, by the induction hypothesis. $\Phi, \Xi \vdash Sys_2\ ok$, by Lemma D.1.1. Thus true for this case.

Therefore the lemma is proved by induction. ■

The following three lemmas state properties of usage disciplines of channels in Φ_{aux} in the translation. Their proofs are straightforward.

Lemma D.1.6 (Disciplines on daemon interfaces)
Given that Φ is a valid system context, if $\Phi \vdash \boldsymbol{A}\ ok$ then the names `register`, `migrating`, and `message` are non-sendable by $\mathcal{F}[\![\boldsymbol{A}]\!]$, and are output-only in $\mathcal{F}[\![\boldsymbol{A}]\!]$, w.r.t. Φ.

Lemma D.1.7 (`currentloc` is a local channel)
Given that Φ is a valid system context, if $\Phi \vdash_a [P\ \boldsymbol{E}]\ ok$ then `currentloc` is a local channel in $\mathcal{F}[\![[P\ \boldsymbol{E}]]\!]_a$.

Lemma D.1.8 (`deliver` is uniformly receptive)
Given that Φ is a valid system context, if $\Phi \vdash \boldsymbol{A}\ ok$ then $\texttt{deliver} : \mathsf{dom}(\boldsymbol{A}) \Vdash \mathcal{F}[\![\boldsymbol{A}]\!]$.

Lemma D.1.9 (Agents are temporarily immobile)
If $Sys = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$ is idle and $\Phi \vdash Sys\ ok$ then

$$\textbf{new}\ \Omega_{aux}\ \textbf{in}\ @_D(Daemon|\mathsf{mesgQ})\ |\ \mathcal{F}[\![\boldsymbol{A}]\!]$$

is temporarily immobile under `lock`, w.r.t. $((\Phi, \Delta, \Omega_D), \emptyset)$.

Proof: Assuming $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ (explicit alpha-conversion if necessary), by the typing rules of IL, $\Phi, \Delta, \Omega_D \vdash \textbf{new}\ \Omega_{aux}\ \textbf{in}\ @_D(Daemon|\mathsf{mesgQ})\ |\ \mathcal{F}[\![\boldsymbol{A}]\!]$. By Lemma D.1.6 and Lemma 6.2.4, we have:

$$\begin{aligned}&\textbf{new}\ \Omega_{aux}\ \textbf{in}\ @_D(Daemon|\mathsf{mesgQ})\ |\ \mathcal{F}[\![\boldsymbol{A}]\!]\\ &\quad\sim_{\Phi,\Delta,\Omega_D}\ \textstyle\prod_{a\in\mathsf{dom}(\boldsymbol{A})}(\textbf{new}\ \Omega_{aux}\ \textbf{in}\ (@_D(Daemon|\mathsf{mesgQ}_a)\ |\ \mathcal{F}[\![\boldsymbol{A}(a)]\!]_a))\end{aligned}$$

where $\mathsf{mesgQ}_a = \mathsf{mesgQ}$ for a unique $a \in \mathsf{dom}(\boldsymbol{A})$ and $\mathbf{0}$ otherwise. By Lemma 6.4.3, if $\boldsymbol{A}(a) \neq [P\ \mathsf{MrdyA}(s\ P)]$ then $(\textbf{new}\ \Omega_{aux}\ \textbf{in}\ (@_D(Daemon|\mathsf{mesgQ}_a)\ |\ \mathcal{F}[\![\boldsymbol{A}(a)]\!]_a))$ is temporary immobile under `lock` w.r.t. $((\Phi, \Delta, \Omega_D), \emptyset)$, for each $a \in \mathsf{dom}(\boldsymbol{A})$. By Lemma 6.4.7 and Lemma 6.4.8, we prove the lemma. ∎

D.2 Analysis of $\mathcal{F}[\![\cdot]\!]$

Lemma D.2.1 (Completeness of IL (7.3.3))
Given a valid system context Φ, if $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ then there exists LP' such that $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow[\Xi]{\beta} LP'$, and $LP' \dot{\succeq}^{\emptyset}_{\Phi,\Xi} \mathcal{F}[\![Sys']\!]$.

Proof: Consider $Sys = \mathsf{eProg}(\Delta'; \boldsymbol{D}'; \boldsymbol{A}')$ well-formed wrt Φ. Since $\mathcal{F}$ is only defined if $\mathsf{dom}(\Delta') \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, we pick an injective substitution $\sigma : \mathsf{dom}(\Delta') \to \mathcal{X}/\mathsf{dom}(\Phi, \Phi_{aux})$ and denote $\Delta = \sigma\Delta'$, $\boldsymbol{D} = \sigma\boldsymbol{D}$ and $\boldsymbol{A}' = \sigma\boldsymbol{A}$. Clearly we have $Sys \stackrel{\alpha}{=} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ and, by (L-C-VAR), $\vdash \Phi, \Phi_{aux}, \Delta$. By this alpha-conversion, $\mathcal{F}[\![Sys]\!]$ is defined.

Supposing $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta) = \emptyset$, the transition must be derived by one of the following cases.

Case (Sys-Loc-IfLocal-T): Supposing $\boldsymbol{A}(a_l) \equiv [P \mid Q\ \boldsymbol{E}]$, $\boldsymbol{A}(b) \equiv [R\ \boldsymbol{E}']$ with $P = \texttt{iflocal}\ \langle b \rangle c!v\ \texttt{then}\ P_1\ \texttt{else}\ P_2$; and $a \neq b$ with $\Phi, \Delta \vdash a_l@s \wedge b@s$. By (Sys-Loc-IfLocal-T), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_l \mapsto [P_1 \mid Q\ \boldsymbol{E}] \oplus b \mapsto [c!v \mid R\ \boldsymbol{E}'])$$

Assuming Sys is idle (the busy case can be handled in the similar way), let $\boldsymbol{A}'$ be obtained by excluding a_l, b from the domain of $\boldsymbol{A}$, we have $\mathcal{F}[\![Sys]\!] \equiv \mathcal{E}[LQ]$, where:

$$\begin{array}{rcl}
\mathcal{E}[\cdot] & = & \texttt{new}\ \Phi_{aux}, \Delta\ \texttt{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \mid [\cdot]) \\
LQ & = & \mathcal{F}[\![\boldsymbol{A}(a_l)]\!]_{a_l} \mid \mathcal{F}[\![\boldsymbol{A}(b)]\!]_b \\
& = & @_{a_l}([\![P \mid Q]\!]_{a_l} \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \mid @_b([\![R]\!]_b \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}']\!]_b \\
& = & @_{a_l}\texttt{iflocal}\ \langle b \rangle c!v\ \texttt{then}\ [\![P_1]\!]_{a_l}\ \texttt{else}\ [\![P_2]\!]_{a_l} \\
& & \quad \mid @_{a_l}([\![Q]\!]_{a_l} \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \mid @_b([\![R]\!]_b \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}']\!]_b
\end{array}$$

In this case, we have:

- $\mathcal{E}[\Phi] \Vdash LQ \xrightarrow{\tau} LQ'$, by (Lts-L-IfLocal-T), (Lts-Prl) and (Lts-New) where

$$\begin{array}{rcl}
LQ' & \equiv & @_{a_l}[\![P_1]\!]_{a_l} \mid @_{a_l}([\![Q]\!]_{a_l} \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \\
& & \mid @_b(c!v \mid [\![R]\!]_b \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}']\!]_b \\
& = & @_{a_l}([\![P_1 \mid Q]\!]_{a_l} \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \\
& & \mid @_b([\![c!v \mid R]\!]_b \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}']\!]_b \\
& = & \mathcal{F}[\![[P_1 \mid Q\ \boldsymbol{E}]]\!]_{a_l} \mid \mathcal{F}[\![[c!v \mid R\ \boldsymbol{E}']]\!]_b
\end{array}$$

- $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow{\tau} \mathcal{E}[LQ']$, by (Lts-Prl) and (Lts-New).

Clearly, $\mathcal{E}[LQ'] \equiv \mathcal{E}[LQ''] \equiv \mathcal{F}[\![Sys']\!]$, so picking $LP' = \mathcal{E}[LQ']$ implies the lemma is true for this case.

Case (Sys-Loc-Tau, Sys-Loc-IfLocal-Same, Sys-Loc-IfLocal-F): Similar to the previous case.

Case (Sys-Loc-Input): Suppose the following

- $\boldsymbol{A}(a_l) \equiv [(c?p \to P) \mid Q\ \boldsymbol{E}]$ and $\{a_l, c\} \subseteq \mathsf{dom}(\Phi)$;
- $\Phi \vdash c \in \hat{\ }^{\mathrm{r}}T$, $\Phi, \Xi \vdash v \in T$ and $\mathsf{dom}(\Xi) \subseteq \mathsf{fv}(v)$ with Ξ is extensible; and

- $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta, \Phi_{aux}) = \emptyset$.

We have, by (SYS-LOC-INPUT), $\Phi \Vdash Sys \xrightarrow[\Xi]{@_{a_l} c?v} Sys'$, where

$$Sys' \;=\; \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_l \mapsto [\mathsf{match}(p, v)P \mid Q\ \boldsymbol{E}])$$

Assuming Sys is idle (the busy case can be handled in the similar way), let $\boldsymbol{A}'$ be obtained by excluding a_l from the domain of $\boldsymbol{A}$, we have $\mathcal{F}[\![Sys]\!] \equiv \mathcal{E}[LQ]$, where:

$$\begin{array}{rcl}
\mathcal{E}[\cdot] & \equiv & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_{\bullet}(\Xi_{map}) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \mid [\cdot]) \\
LQ & = & \mathcal{F}[\![\boldsymbol{A}(a_l)]\!] \\
 & = & @_{a_l}([\![(c?p \to P) \mid Q]\!]_{a_l} \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \\
 & = & @_{a_l} c?p \to [\![P]\!]_{a_l} \mid @_{a_l}([\![Q]\!]_{a_l} \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l}
\end{array}$$

In this case, we have:

- $\Phi, \Xi \vdash v \in T$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta, \Phi_{aux}) = \emptyset$ implies $\mathcal{E}[\Phi], \Xi \vdash v \in T$, by Lemma 3.7.2. This means that $\mathcal{E}[\Phi] \Vdash LQ \xrightarrow[\Xi]{@_{a_l} c?v} LQ'$, by (LTS-L-IN), (LTS-PRL) and (LTS-NEW) where

$$LQ' \;\equiv\; @_{a_l} \mathsf{match}(p, v)\, [\![P]\!]_{a_l} \mid @_{a_l}([\![Q]\!]_{a_l} \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l}$$

- Since $\mathsf{match}(p, v)\, [\![P]\!] = [\![\mathsf{match}(p, v)P]\!]$ (as $p \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$), we have

$$\begin{array}{rcl}
LQ' & \equiv & @_{a_l} [\![\mathsf{match}(p, v)P]\!] \mid @_{a_l}([\![Q]\!] \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \\
 & = & \mathcal{F}[\![[\mathsf{match}(p, v)P \mid Q\ \boldsymbol{E}]]\!]_{a_l}
\end{array}$$

- $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow[\Xi]{@_{a_l} c?v} \mathcal{E}[LQ']$, since $\mathsf{fv}(v) \cap \mathsf{dom}(\Phi_{aux}, \Delta) = \emptyset$ and apply (LTS-PRL) and (LTS-NEW).

Clearly, $\mathcal{E}[LQ'] \equiv \mathcal{F}[\![Sys']\!]$, picking $LP' = \mathcal{E}[LQ']$ implies the lemma is true for this case.

Case (Sys-Loc-Replic): Similar to the previous case.

Case (Sys-Loc-Output): Suppose $\boldsymbol{A}(a_l) \equiv [c!v \mid Q\ \boldsymbol{E}]$, $\{a_l, c\} \subseteq \mathsf{dom}(\Phi)$, and $\Delta \equiv \Delta_1, \Delta_2$ with $\mathsf{dom}(\Delta) \cap \mathsf{fv}(v) = \mathsf{dom}(\Delta_1)$. We have, by (SYS-LOC-OUTPUT), $\Phi \Vdash Sys \xrightarrow[\Delta_1]{@_{a_l} c!v} Sys'$, where

$$Sys' \;=\; \mathsf{eProg}(\Delta_2; \boldsymbol{D}; \boldsymbol{A} \oplus a_l \mapsto [Q\ \boldsymbol{E}])$$

Assuming Sys is idle (the busy case can be handled in the similar way), let $\boldsymbol{A}'$ be obtained by excluding a_l from the domain of $\boldsymbol{A}$, we have $\mathcal{F}[\![Sys]\!] \equiv \mathcal{E}[LQ]$, where:

$$\begin{array}{rcl} \mathcal{E}[\cdot] & \equiv & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(\Xi_{map}) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \mid [\cdot]) \\ LQ & = & \mathcal{F}[\![\boldsymbol{A}(a_l)]\!] \\ & = & @_{a_l}([\![c!v \mid Q]\!]_{a_l} \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \\ & = & @_{a_l}c!v \mid @_{a_l}([\![Q]\!]_{a_l} \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l} \end{array}$$

In this case, we have:

- $\mathcal{E}[\Phi] \Vdash LQ \xrightarrow{@_{a_l}c!v} LQ'$, by (Lts-L-Out) and (Lts-Prl) where
$$LQ' \equiv @_{a_l}([\![Q]\!]_{a_l} \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_l}$$
- $\Phi \vdash Sys\ ok$ implies (by various typing rules), $\mathsf{fv}(v) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, ie. neither `currentloc` nor names in Φ_{aux} is extruded by this outputting.
- $\Phi \Vdash LP \xrightarrow[\Delta_1]{@_{a_l}c!v} \mathcal{E}'[LQ']$, by (Lts-Open), (Lts-Prl) and (Lts-New), where $\mathcal{E}'[\cdot]$ is defined as below.

$$\mathbf{new}\ \Phi_{aux}, \Delta_2\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(\Xi_{map}) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \mid [\cdot])$$

Clearly, $\mathcal{E}'[LQ'] \equiv \mathcal{F}[\![Sys']\!]$, pick $LP' \equiv \mathcal{E}'[LQ']$ implies the lemma is true for this case.

Case (Sys-Req-Reg): Supposing $\boldsymbol{A}(a_e) \equiv [(\mathbf{create}^Z\ b' = P'\ \mathbf{in}\ Q') \mid R\ \mathsf{FreeA}(s)]$ and $b' \notin \mathsf{dom}(\Phi, \Delta)$. For avoiding name clashes, we pick an injective substitution $\rho : \{b'\} \to \mathcal{X}/\mathsf{dom}(\Phi, \Delta, \Phi_{aux})$ and write $b = \rho b'$, $P = \rho P'$ and $Q = \rho Q'$. This means $\vdash \Phi, \Delta, \Phi_{aux}, b : \mathtt{Agent}^Z @s$, by (L-C-Var). Clearly, $Sys \stackrel{\alpha}{=} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}_\rho)$ where $\boldsymbol{A}_\rho$ is obtained by alpha-converting the name b' and hence: $\boldsymbol{A}_\rho(a_e) = [(\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q) \mid R\ \mathsf{FreeA}(s)]$. In this case, we have, by (Sys-Req-Reg), $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where

$$Sys' = \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)])$$

Assume (wlog.) that Sys is idle (the case where Sys is busy can be derived similarly). Let $\boldsymbol{A}'$ be obtained by excluding a_e from the domain of $\boldsymbol{A}_\rho$, and $\mathcal{E}[\cdot], LQ$ be defined as follows:

$$\begin{array}{rcl} \mathcal{E}[\cdot] & = & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \mid [\cdot]) \\ LQ & = & \mathcal{F}[\![\boldsymbol{A}_\rho(a_e)]\!] \\ & = & @_{a_e}\left([\![(\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q) \mid R]\!]_{a_e} \mid \mathtt{Deliverer}\right) \mid \mathcal{F}[\![\mathsf{FreeA}(s)]\!]_{a_e} \\ & = & @_{a_e}\left([\![\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q]\!]_{a_e} \mid [\![R]\!]_{a_e} \mid \mathtt{Deliverer} \mid \mathtt{currentloc!}s\right) \end{array}$$

Clearly $LP = \mathcal{F}[\![Sys]\!] \equiv \mathcal{E}[LQ]$. In this case, we have:

- Assuming that pack, rack are fresh names, $\mathcal{E}[\Phi] \Vdash LQ \xrightarrow{\tau} LQ'$, by (LTS-L-OUT), (LTS-L-IN), (LTS-L-COMM), and (LTS-PRL), where LQ' is structurally congruent to the process below.

$$\begin{array}{l} @_{a_e}\mathbf{new}\ \mathtt{pack} : \mathtt{\char94 rw}[], \mathtt{rack} : \mathtt{\char94 rw}[]\ \mathbf{in} \\ \quad (\mathbf{create}^Z\ b = \langle D@SD\rangle \mathtt{register!}[b\ s\ \mathtt{rack}] \mid \mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ s\ P) \\ \quad \mathbf{in}\ \mathsf{regBlockP}(\mathtt{pack}\ s\ Q)) \\ \mid @_{a_e}([\![R]\!] \mid \mathtt{Deliverer}) \end{array}$$

- $\Phi \Vdash LP \xrightarrow{\tau} LP'$, where $LP' = \mathcal{E}[LQ']$ by (LTS-PRL) and (LTS-NEW).

We now need to verify that $LP' \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{F}[\![Sys']\!]$. Let $M = \mathsf{mov}(\mathcal{E}[\Phi])/\{a_e\}$, we have:

- Since $\Phi, \Delta \vdash a_e@s$ (by (SYS-T-FREEA)), applying Lemma 6.3.4, we have $\mathcal{E}[\Phi] \Vdash LR_1 \xrightarrow[M]{\mathsf{det}} LR_2$ where

$$\begin{array}{rcl} LR_1 & = & \mathbf{new}\ \mathtt{pack} : \mathtt{\char94 rw}[], \mathtt{rack} : \mathtt{\char94 rw}[]\ \mathbf{in} \\ & & \quad (@_{a_e}\mathbf{create}^Z\ b = \\ & & \qquad \langle D@SD\rangle \mathtt{register!}[b\ s\ \mathtt{rack}] \mid \mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ s\ P) \\ & & \quad\ \mathbf{in}\ \mathsf{regBlockP}(\mathtt{pack}\ s\ Q)) \\ LR_2 & = & \mathbf{new}\ \mathtt{pack} : \mathtt{\char94 rw}[], \mathtt{rack} : \mathtt{\char94 rw}[], b : \mathtt{Agent}^Z@s\ \mathbf{in} \\ & & \quad (@_{a_e}\mathsf{regBlockP}(\mathtt{pack}\ s\ Q) \\ & & \quad \mid @_b(\langle D@SD\rangle \mathtt{register!}[b\ s\ \mathtt{rack}] \mid \mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ s\ P))) \end{array}$$

- We need to redefine the scope of currentloc, in order to use the temporary immobility result. By Lemma D.1.7 and applying Lemma 6.2.7, we have

$$\begin{array}{rcl} \mathcal{E}[LQ'] & \dot{\sim}_{\Phi} & \mathcal{E}'[\mathbf{new}\ \mathtt{currentloc} : \mathtt{\char94 rw}\mathtt{Site}\ \mathbf{in}\ LQ'] \quad \text{where} \\ \mathcal{E}'[\cdot] & = & \mathbf{new}\ \Phi^-_{aux}, \Delta\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(map) \mid [\cdot] \\ & & \qquad \mid \mathbf{new}\ \mathtt{currentloc} : \mathtt{\char94 rw}\mathtt{Site}\ \mathbf{in}\ \mathcal{F}[\![\boldsymbol{A'}]\!]) \end{array}$$

- By Lemma 6.4.4, $@_{a_e}([\![R]\!] \mid \mathtt{Deliverer})$ is temporarily immobile, under currentloc, wrt $(\mathcal{E}[\Phi], M)$, applying Lemma 6.4.2, we have:

$$\mathbf{new}\ \mathtt{currentloc} : \mathtt{\char94 rw}\mathtt{Site}\ \mathbf{in}\ LQ' \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} \mathbf{new}\ \mathtt{currentloc} : \mathtt{\char94 rw}\mathtt{Site}\ \mathbf{in}\ LQ_1$$

where

$$\begin{array}{rcl} LQ_1 & = & LR_2 \mid @_{a_e}([\![R]\!] \mid \mathtt{Deliverer}) \\ & \equiv & \mathbf{new}\ \mathtt{pack} : \mathtt{\char94 rw}[], \mathtt{rack} : \mathtt{\char94 rw}[], b : \mathtt{Agent}^Z@s\ \mathbf{in} \\ & & \quad @_{a_e}(\mathsf{regBlockP}(\mathtt{pack}\ s\ Q) \mid [\![R]\!] \mid \mathtt{Deliverer}) \\ & & \quad \mid @_b(\langle D@SD\rangle \mathtt{register!}[b\ s\ \mathtt{rack}] \mid \mathsf{regBlockC}(\mathtt{rack}\ \mathtt{pack}\ s\ P)) \end{array}$$

- Now that the temporary immobility is applied, we may now use the "normal" scoping for `currentloc`. By Lemma D.1.7 and applying Lemma 6.2.7, we have

$$\mathcal{E}'[\textbf{new}\ \texttt{currentloc} : \texttt{\^{}rwSite}\ \textbf{in}\ LQ_1] \quad \dot{\sim}_\Phi \quad \mathcal{E}[LQ_1]$$

- Since D is a static agent at SD (ie. $D \notin \mathsf{mov}(\mathcal{E}[\Phi], b : \texttt{Agent}^Z@s))$, by Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), we have:

$$@_b\langle D@SD\rangle\texttt{register!}[b\ s\ \texttt{rack}] \quad \succeq_{\mathcal{E}[\Phi],b:\texttt{Agent}^Z@s,\texttt{rack}:\texttt{\^{}rw}[]} \quad @_D\texttt{register!}[b\ s\ \texttt{rack}]$$

- By Theorem 5.4.3 (cong property for expansion), we have $LQ_1 \dot{\succeq}^M_{\mathcal{E}[\Phi]} LQ_2$, where:

$$\begin{array}{rcl} LQ_2 & \equiv & \textbf{new}\ b : \texttt{Agent}^Z@s, \texttt{pack} : \texttt{\^{}rw}[], \texttt{rack} : \texttt{\^{}rw}[]\ \textbf{in}\ (@_D\texttt{register!}[b\ s\ \texttt{rack}] \\ & & \quad \mid @_{a_e}(\mathsf{regBlockP}(\texttt{pack}\ s\ Q) \mid [\![R]\!] \mid \texttt{Deliverer}) \\ & & \quad \mid @_b\mathsf{regBlockC}(\texttt{rack}\ \texttt{pack}\ s\ P)) \end{array}$$

- By transitivity of expansion relation, $LQ' \dot{\succeq}^M_{\mathcal{E}[\Phi]} LQ_2$. By Lemma D.1.3, $a_e \notin \mathsf{mayMove}(@_D Daemon \mid \mathsf{mesgQ} \mid \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}']\!])$, applying Theorem 5.4.3 (cong property for expansion), we have: $\mathcal{E}[LQ'] \dot{\succeq}^\emptyset_\Phi \mathcal{E}[LQ_2]$.

- Since, by Lemma D.1.6, `register` is handled by LQ_2, $\mathcal{F}[\![\boldsymbol{A}']\!]$ and $\mathsf{mapS}_\bullet(map)$, applying Lemma 6.2.3, we have: $\mathcal{E}[LQ_2] \dot{\succeq}^\emptyset_\Phi LP''$ where:

$$\begin{array}{rcl} LP'' & \equiv & \textbf{new}\ \Phi_{aux}, \Delta\ \textbf{in} \\ & & \quad (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \\ & & \quad \mid \textbf{new}\ b : \texttt{Agent}^Z@s, \texttt{pack} : \texttt{\^{}rw}[], \texttt{rack} : \texttt{\^{}rw}[]\ \textbf{in}\ @_D\mathsf{regReq}(b\ s\ \texttt{rack}) \\ & & \qquad \mid @_{a_e}(\mathsf{regBlockP}(\texttt{pack}\ s\ Q) \mid [\![R]\!] \mid \texttt{Deliverer}) \\ & & \qquad \mid @_b\mathsf{regBlockC}(\texttt{rack}\ \texttt{pack}\ s\ P)) \\ & = & \textbf{new}\ \Phi_{aux}, \Delta\ \textbf{in} \\ & & \quad (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \\ & & \quad \mid \mathcal{F}[\![[R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)]]\!]_{a_e}) \end{array}$$

Clearly $LP'' \equiv \mathcal{F}[\![Sys']\!]$ and hence $LP' \dot{\succeq}^\emptyset_\Phi \mathcal{F}[\![Sys']\!]$, by transitivity of expansion relation. The lemma is true for this case.

Case (Sys-Req-Mig, Sys-Req-Mesg): Similar to the previous case, although considerably simpler: we neither need alpha-conversion nor the temporaly immobility result. We omit their details.

Case (Sys-Proc-Reg): Supposing $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ is idle with $\boldsymbol{D} = [map\ \mathsf{mesgQ}]$, $\boldsymbol{A}(a_e) = [R\ \mathsf{RegA}(b'\ Z\ s\ P'\ Q')]$, and $b' \notin \mathsf{dom}(\Phi, \Delta)$. In order that $\mathcal{F}[\![Sys]\!]$ is defined, we pick an injective substitution $\rho : \{b'\} \to \mathcal{X}/\mathsf{dom}(\Phi, \Delta, \Phi_{aux})$ and write $b = \rho b'$, $P = \rho P'$ and $Q = \rho Q'$. This means $\vdash \Phi, \Delta, \Phi_{aux}, b : \mathtt{Agent}^Z@s$ (by (L-C-VAR)) and $Sys \stackrel{\alpha}{=} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A}_\rho)$ where $\boldsymbol{A}_\rho$ is obtained by alpha-converting the name b' (ie. $\boldsymbol{A}_\rho(a_e) = [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)]$). In this case, we have, by (SYS-PROC-REG) (and (SYS-EQUIV)), $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where Sys' denotes the system below.

$$\mathsf{eProg}(\Delta, b : \mathtt{Agent}^Z@s; [[b\ s]\mathbf{::}map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [Q|R\ \mathsf{FreeA}(s)] \oplus b \mapsto [P\ \mathsf{FreeA}(s)])$$

Let $\boldsymbol{A}'$ be obtained by excluding a_e from the domain of $\boldsymbol{A}_\rho$ and $\mathcal{E}[\cdot]$, LQ, LR be defined as follows.

$$\begin{array}{rcl}
\mathcal{E}[\cdot] & = & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid [\cdot] \mid \mathcal{F}[\![\boldsymbol{A}']\!]) \\
LQ & = & \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}_\rho(a_e)]\!]_{a_e} \\
 & = & @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathtt{lock!}m \mid \mathsf{makeMap}(m; map)) \\
 & & \mid @_{a_e}([\![R]\!]_{a_e} | \mathtt{Deliverer}) \mid \mathcal{F}[\![\mathsf{RegA}(b\ Z\ s\ P\ Q)]\!]_{a_e} \\
 & = & @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathtt{lock!}m \mid \mathsf{makeMap}(m; map)) \\
 & & \mid @_{a_e}([\![R]\!]_{a_e} | \mathtt{Deliverer}) \\
 & & \mid \mathbf{new}\ \mathtt{pack} : \texttt{\^{}rw}[], \mathtt{rack} : \texttt{\^{}rw}[], b : \mathtt{Agent}^Z@s\ \mathbf{in}\ (@_D\mathsf{regReq}(b\ s\ \mathtt{rack}) \\
 & & \quad \mid @_b\mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P) \mid @_{a_e}\mathsf{regBlockP}(s\ \mathtt{pack}\ Q)) \\
 & \equiv & \mathbf{new}\ \mathtt{pack} : \texttt{\^{}rw}[], \mathtt{rack} : \texttt{\^{}rw}[], b : \mathtt{Agent}^Z@s\ \mathbf{in}\ (LR \\
 & & \quad \mid @_D(\mathsf{regReq}(b\ s\ \mathtt{rack}) \\
 & & \qquad \mid \mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathtt{lock!}m \mid \mathsf{makeMap}(m; map)))) \\
LR & = & @_{a_e}(\mathsf{regBlockP}(s\ \mathtt{pack}\ Q) \mid [\![R]\!]_{a_e} \mid \mathtt{Deliverer}) \mid @_b\mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P)
\end{array}$$

Clearly $\mathcal{F}[\![Sys]\!] \equiv \mathcal{E}[LQ]$. In this case, we have:

- $\mathcal{E}[\Phi] \Vdash LQ \xrightarrow{\tau} LQ'$, by (LTS-L-OUT), (LTS-L-IN) and (LTS-L-COMM), (LTS-PRL) and (LTS-NEW), where LQ' is the process below.

$$\begin{array}{rcl}
LQ' & \equiv & \mathbf{new}\ \mathtt{pack} : \texttt{\^{}rw}[], \mathtt{rack} : \texttt{\^{}rw}[], b : \mathtt{Agent}^Z@s\ \mathbf{in}\ LR \\
 & & \mid @_D(\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathsf{makeMap}(m; map) \\
 & & \quad \mid \mathbf{let}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ m' = (m\ \mathbf{with}\ b \mapsto s)\ \mathbf{in} \\
 & & \qquad (\mathtt{lock!}m' \mid \langle b@s \rangle \mathtt{rack!}[])))
\end{array}$$

- Pick $LP' = \mathcal{E}[LQ']$, we have $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow{\tau} LP'$, by (LTS-PRL) and (LTS-NEW).

We now need to verify that $LP' \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{F}[\![Sys']\!]$. Let $M = \mathsf{mov}(\mathcal{E}[\Phi])/\{a_e\}$.

- By Lemma 6.5.2, we have: $LQ' \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} LQ_1$ where

$$\begin{array}{rcl} LQ_1 & = & \mathbf{new}\ \mathtt{pack}, \mathtt{rack} : \mathtt{\hat{}rw}[], b : \mathtt{Agent}^Z @ s\ \mathbf{in}\ (LR \mid @_D \langle b@s \rangle \mathtt{rack!}[] \\ & & \quad \mid @_D \mathbf{new}\ m' : \mathtt{Map}[\mathtt{Agent^s\ Site}]\ \mathbf{in} \\ & & \qquad (\mathsf{makeMap}(m'; [b\ s] \mathtt{::} map) \mid \mathtt{lock!} m')) \\ & = & \mathbf{new}\ \mathtt{pack}, \mathtt{rack} : \mathtt{\hat{}rw}[], b : \mathtt{Agent}^Z @ s\ \mathbf{in}\ (LR \mid @_D \langle b@s \rangle \mathtt{rack!}[] \\ & & \quad \mid \mathsf{mapS}_{\bullet}([b\ s] \mathtt{::} map)) \end{array}$$

- Since $\mathcal{E}[\Phi], b : \mathtt{Agent}^Z @ s \vdash b@s$, by Lemma 6.3.5, we have:

$$\Phi' \Vdash @_D \langle b@s \rangle \mathtt{rack!}[] \xrightarrow[\mathsf{mov}(\mathcal{E}[\Phi])]{\mathsf{det}} @_b \mathtt{rack!}[]$$

where $\Phi' = \mathcal{E}[\Phi], \mathtt{rack} : \mathtt{\hat{}rw}[], \mathtt{pack} : \mathtt{\hat{}rw}[], b : \mathtt{Agent}^Z @ s$. By Lemma 6.4.5,

$$@_b \mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P)$$

is obviously temporary immobile under $\mathtt{rack}$, wrt $(\Phi', \mathsf{mov}(\Phi')/\{b\})$, applying Lemma 6.4.2, we have:

$$\begin{array}{ll} & \mathbf{new}\ \mathtt{rack} : \mathtt{\hat{}rw}[]\ \mathbf{in}\ (@_D \langle b@s \rangle \mathtt{rack!}[] \mid @_b \mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P)) \\ \dot{\succeq}^{\mathsf{mov}(\Phi')/\{b\}}_{\Phi'/\mathtt{rack}:\mathtt{\hat{}rw}[]} & \mathbf{new}\ \mathtt{rack} : \mathtt{\hat{}rw}[]\ \mathbf{in}\ (@_b \mathtt{rack!}[] \mid @_b \mathsf{regBlockC}(s\ \mathtt{pack}\ \mathtt{rack}\ P)) \\ \dot{\succeq}^{\mathsf{mov}(\Phi')/\{b\}}_{\Phi'/\mathtt{rack}:\mathtt{\hat{}rw}[]} & @_b \mathbf{iflocal}\ \langle a_e \rangle \mathtt{pack!}[]\ \mathbf{then}\ (\mathtt{currentloc!} s \mid [\![P]\!]_b \mid \mathtt{Deliverer}) \end{array}$$

The latter expansion is derived using Lemma 6.3.7 and using structural congruence to get rid of rack-binding.

- Since $b \notin \mathsf{mayMove}(\mathsf{mapS}_{\bullet}([b\ s] \mathtt{::} map), @_{a_e} \ldots)$, applying Theorem 5.4.3 (cong property for expansion), we have $LQ_1 \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} LQ_2$, where LQ_2 denotes the process below.

$$\begin{array}{l} \mathbf{new}\ \mathtt{pack} : \mathtt{\hat{}rw}[], b : \mathtt{Agent}^Z @ s\ \mathbf{in}\ \mathsf{mapS}_{\bullet}([b\ s] \mathtt{::} map) \\ \quad \mid @_b \mathbf{iflocal}\ \langle a_e \rangle \mathtt{pack!}[]\ \mathbf{then}\ (\mathtt{currentloc!} s \mid [\![P]\!]_b \mid \mathtt{Deliverer}) \\ \quad \mid @_{a_e}(\mathsf{regBlockP}(s\ \mathtt{pack}\ Q) \mid [\![R]\!]_{a_e} \mid \mathtt{Deliverer}) \end{array}$$

- By Lemma 5.3.1, $LQ' \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} LQ_2$. By Lemma D.1.3, we have

$$a_e \notin \mathsf{mayMove}(@_D(Daemon \mid \mathsf{mesgQ}), \mathcal{F}[\![\boldsymbol{A}']\!])$$

Applying Theorem 5.4.3 (cong property for expansion), we have $\mathcal{E}[LQ'] \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{E}[LQ_2]$.

For the next part of this case, we shall focus on the interaction between parent and child agents. We now define $\mathcal{E}'[\cdot]$ as follows.

$$\begin{array}{rcl}\mathcal{E}'[\cdot] & = & \textbf{new}\ \Phi^{-}_{aux}, \Delta, b : \texttt{Agent}^{Z}@s\ \textbf{in} \\ & & \quad (@_D(\mathit{Daemon} \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_{\bullet}([b\ s]\texttt{::}\mathit{map}) \mid [\cdot] \\ & & \quad \mid \textbf{new}\ \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\ \textbf{in}\ \mathcal{F}[\![\boldsymbol{A'}]\!]) \\ \Phi^{-}_{aux} & \stackrel{\text{def}}{=} & \Phi_{aux}/\texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\end{array}$$

The redefined scope of `currentloc` will take care of temporary immobility of a_e. By Lemma D.1.7 and Lemma 6.2.7, $\mathcal{E}[LQ_2]\dot{\sim}^{\emptyset}_{\Phi}\mathcal{E}'[LR_1]$, where LR_1 is defined as the process below.

$$\begin{array}{rcl}LR_1 & = & \textbf{new}\ \texttt{pack} : \texttt{\^{}}^{\texttt{rw}}[]\ \textbf{in}\ (LR_p \mid LR_c) \\ LR_c & = & \textbf{new}\ \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\ \textbf{in} \\ & & \quad (@_b\textbf{iflocal}\ \langle a_e\rangle\texttt{pack!}[]\ \textbf{then}\ (\texttt{currentloc!}s \mid [\![P]\!]_b \mid \texttt{Deliverer})) \\ LR_p & = & \textbf{new}\ \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\ \textbf{in} \\ & & \quad (@_{a_e}([\![R]\!]_{a_e} \mid \mathsf{regBlockP}(s\ \texttt{pack}\ Q) \mid \texttt{Deliverer}))\end{array}$$

Let $M' = \mathsf{mov}(\Phi)/\{a_e, b\}$, we have:

- Since, by Lemma 6.4.5, LR_p is temporary immobile under `pack` wrt $(\mathcal{E}[\Phi], M)$. By Lemma 6.3.3 and (Sys-T-RegA), $\mathcal{E}'[\Phi] \vdash a@s \wedge b@s$, this implies

  $$\begin{array}{l}\mathcal{E}'[\Phi] \Vdash LR_c \xrightarrow[M']{\mathsf{det}} LR'_c \qquad \text{where} \\ \qquad LR'_c \quad = \quad \textbf{new}\ \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\ \textbf{in} \\ \qquad\qquad\qquad @_{a_e}\texttt{pack!}[] \mid @_b(\texttt{currentloc!}s \mid [\![P]\!]_b \mid \texttt{Deliverer})\end{array}$$

  Applying Lemma 6.4.2, we have $LR_1 \dot{\succeq}^{M'}_{\mathcal{E}[\Phi]} LR_2$, where

  $$LR_2 \equiv \textbf{new}\ \texttt{pack} : \texttt{\^{}}^{\texttt{rw}}[]\ \textbf{in}\ (LR_p|LR'_c)$$

- By Lemma 6.3.7, we have $LR_2 \dot{\succeq}^{M'}_{\mathcal{E}'[\Phi]} LR_3$ where

  $$\begin{array}{rcl}LR_3 & \equiv & \textbf{new}\ \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\ \textbf{in} \\ & & \quad (@_b(\texttt{currentloc!}s \mid [\![P]\!]_b \mid \texttt{Deliverer})) \\ & & \mid \textbf{new}\ \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\ \textbf{in} \\ & & \quad (@_{a_e}([\![R]\!]_{a_e} \mid [\![Q]\!]_{a_e} \mid \texttt{currentloc!}s \mid \texttt{Deliverer}))\end{array}$$

- By Lemma D.1.7 and Lemma 6.2.7, $LR_3 \dot{\sim}^{\emptyset}_{\Phi} LR_4$ where

  $$LR_4 = \textbf{new}\ \texttt{currentloc} : \texttt{\^{}}^{\texttt{rw}}\texttt{Site}\ \textbf{in}\ (\mathcal{F}[\![[P\ \mathsf{FreeA}(s)]]\!]_b \mid \mathcal{F}[\![[Q|R\ \mathsf{FreeA}(s)]]\!]_{a_e})$$

- $LR \dot{\succeq}_{\Phi}^{M'} LR_4$, by transitivity of expansion relation. By Lemma D.1.3,

$$a_e, b \notin \mathsf{mayMove}(@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_{\bullet}([b\ s]\mathbin{::}map) \mid \mathcal{F}[\![\boldsymbol{A}']\!])$$

  Applying Theorem 5.4.3 (cong property for expansion), we have $\mathcal{E}'[LR] \dot{\succeq}_{\Phi}^{\emptyset} \mathcal{E}'[LR_4]$.

- By Lemma D.1.7 and Lemma 6.2.7, $\mathcal{E}'[LR_4] \dot{\sim}_{\Phi}^{\emptyset} LP''$ where LP'' is as follows.

$$\begin{array}{rcl} LP'' & \equiv & \textbf{new}\ \Phi_{aux}, \Delta, b : \mathtt{Agent}^Z@s\ \textbf{in} \\ & & (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_{\bullet}([b\ s]\mathbin{::}map) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \\ & & \mid \mathcal{F}[\![[P\ \mathsf{FreeA}(s)]]\!]_b \mid \mathcal{F}[\![[Q|R\ \mathsf{FreeA}(s)]]\!]_{a_e}) \end{array}$$

Clearly, $\mathcal{E}[LR''] \equiv \mathcal{F}[\![Sys']\!]$ and $LP' \dot{\succeq}_{\Phi}^{\emptyset} LP''$, by transitivity of expansion relation. Thus the lemma is true for this case.

Case (Sys-Proc-Mig): Given in Section 7.3.3.

Case (Sys-Comm-Mig): Supposing $\boldsymbol{A}(a_e) = [Q\ \mathsf{MrdyA}(s\ P)]$, we have, by (Sys-Proc-Mig-Comm-B), $\Phi\delta \Vdash Sys \xrightarrow{\tau} Sys'$, where

$$Sys' \ = \ \mathsf{eProg}(\Delta \oplus a_e \mapsto s; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [Q|P\ \mathsf{FreeA}(s)])$$

Let $\boldsymbol{A}'$ be obtained by excluding a_e from the domain of $\boldsymbol{A}$ and define $\mathcal{E}[\cdot], \mathcal{E}'[\cdot], LQ$ as follows.

$$\begin{array}{rcl} \mathcal{E}[\cdot] & = & \textbf{new}\ \Phi_{aux}, \Delta\ \textbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid [\cdot] \mid \mathcal{F}[\![\boldsymbol{A}']\!]) \\ \mathcal{E}'[\cdot] & = & \textbf{new}\ \Phi_{aux}, (\Delta \oplus a_e \mapsto s)\ \textbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid [\cdot] \mid \mathcal{F}[\![\boldsymbol{A}']\!]) \\ LQ & = & \mathsf{mapS}_{(a_e, \boldsymbol{A}(a_e))}(map) \\ & = & \textbf{new}\ \mathtt{migrated} : \mathtt{\hat{}^{rw}Site}\ \mathtt{\hat{}^{rw}}[]\ \textbf{in} \\ & & \quad (@_D\textbf{new}\ m : \mathtt{Map[Agent^s\ Site]}\ \textbf{in} \\ & & \qquad (\mathsf{makeMap}(m; map) \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})) \\ & & \quad \mid @_{a_e}(\mathsf{migReady}(s\ \mathtt{migrated}\ P) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer})) \end{array}$$

Clearly, $\mathcal{F}[\![Sys]\!] \equiv \mathcal{E}[LQ]$. In this case, we have:

- $\mathcal{E}[\Phi\delta] \Vdash LQ \xrightarrow{@_a \mathsf{migrate\ to}\ s} LQ'$, by (Lts-L-Mig) and applying (Lts-Prl) and (Lts-New), where:

$$\begin{array}{rcl} LQ' & = & \textbf{new}\ \mathtt{migrated} : \mathtt{\hat{}^{rw}Site}\ \mathtt{\hat{}^{rw}}[]\ \textbf{in} \\ & & \quad (@_D\textbf{new}\ m : \mathtt{Map[Agent^s\ Site]}\ \textbf{in} \\ & & \qquad (\mathsf{makeMap}(m; map) \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})) \\ & & \quad \mid \textbf{new}\ \mathtt{ack} : \mathtt{\hat{}^{rw}}[]\ \textbf{in}\ (@_{a_e}\langle D@SD\rangle \mathtt{migrated!}[s\ \mathtt{ack}] \mid LR)) \\ LR & = & @_{a_e}(\mathtt{ack?}[] \rightarrow (\mathtt{currentloc!}s| [\![P]\!]_{a_e}) \mid [\![Q]\!]_{a_e} \mid \mathtt{Deliverer}) \end{array}$$

- $\Phi\delta \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow{\tau} LP'$, by (Lts-Prl), (Lts-Bound-Mig) and (Lts-New), where $LP' = \mathcal{E}'[LQ']$.

We now need to prove that $LP' \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{F}[\![Sys']\!]$. Let $M = \mathsf{mov}(\mathcal{E}'[\Phi])/\{a_e\}$, we have:

- $@_{a_e}\langle D@SD\rangle\mathtt{migrated!}[s\ \mathtt{ack}] \succeq_{\Phi'} @_D\mathtt{migrated!}[s\ \mathtt{ack}]$, by Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), where $\Phi' = \mathcal{E}'[\Phi], \mathtt{migrated} : \hat{}^{\mathtt{rw}}\mathtt{Site}\ \hat{}^{\mathtt{rw}}[], \mathtt{ack} : \hat{}^{\mathtt{rw}}[]$ (since D is a static agent). This implies, by Theorem 5.4.4 (Transloc Bisim Cong), that $LQ' \dot{\succeq}^{M}_{\mathcal{E}'[\Phi]} LQ_1$ where

$$
\begin{array}{rcl}
LQ_1 & \equiv & \mathbf{new}\ \mathtt{migrated} : \hat{}^{\mathtt{rw}}\mathtt{Site}\ \hat{}^{\mathtt{rw}}[], \mathtt{ack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \\
& & \quad (@_D\mathtt{migrated!}[s\ \mathtt{ack}] \\
& & \quad \mid @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent^s\ Site}]\ \mathbf{in} \\
& & \qquad\quad (\mathsf{makeMap}(m; map) \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})) \\
& & \quad \mid LR) \\
& \equiv & \mathbf{new}\ \mathtt{ack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \\
& & \quad (LR \\
& & \quad \mid \mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent^s\ Site}]\ \mathbf{in}\ (@_D\mathsf{makeMap}(m; map) \mid \\
& & \qquad @_D\mathbf{new}\ \mathtt{migrated} : \hat{}^{\mathtt{rw}}\mathtt{Site}\ \hat{}^{\mathtt{rw}}[]\ \mathbf{in} \\
& & \qquad\quad (\mathtt{migrated!}[s\ \mathtt{ack}] \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated}))))
\end{array}
$$

- By Lemma 6.3.7, we have

$$
\begin{array}{l}
@_D\mathbf{new}\ \mathtt{migrated} : \hat{}^{\mathtt{rw}}\mathtt{Site}\ \hat{}^{\mathtt{rw}}[]\ \mathbf{in}\ (\mathtt{migrated!}[s\ \mathtt{ack}] \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})) \\
\quad \succeq_{\mathcal{E}'[\Phi], \mathtt{ack}:\hat{}^{\mathtt{rw}}[], m:\mathtt{Map}[\mathtt{Agent^s\ Site}]} \\
\qquad @_D\mathbf{let}[\mathtt{Agent^s\ Site}]\ m' = (m\ \mathbf{with}\ a_e \mapsto s)\ \mathbf{in}\ (\mathtt{lock!}m' \mid \langle a_e@s\rangle\mathtt{ack!}[])
\end{array}
$$

This means, by Theorem 5.4.4 (Transloc Bisim Cong), $LQ_1 \dot{\succeq}^{M}_{\mathcal{E}'[\Phi]} LQ_2$ where

$$
\begin{array}{rcl}
LQ_2 & \equiv & \mathbf{new}\ \mathtt{ack} : \hat{}^{\mathtt{rw}}[]\ \mathbf{in}\ (LR \\
& & \quad \mid \mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent^s\ Site}]\ \mathbf{in}\ @_D(\mathsf{makeMap}(m; map) \mid \\
& & \qquad \mathbf{let}[\mathtt{Agent^s\ Site}]\ m' = (m\ \mathbf{with}\ a_e \mapsto s)\ \mathbf{in} \\
& & \qquad\quad (\mathtt{lock!}m' \mid \langle a_e@s\rangle\mathtt{ack!}[])))
\end{array}
$$

- By Lemma 6.5.2, we have

$$
\begin{array}{l}
\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent^s\ Site}]\ \mathbf{in}\ (@_D(\mathsf{makeMap}(m; map) \mid \\
\qquad \mathbf{let}[\mathtt{Agent^s\ Site}]\ m' = (m\ \mathbf{with}\ a_e \mapsto s)\ \mathbf{in}\ \mathtt{lock!}m' \mid \langle a_e@s\rangle\mathtt{ack!}[])) \\
\quad \succeq_{\mathcal{E}'[\Phi], \mathtt{ack}:\hat{}^{\mathtt{rw}}[]}\ \mathbf{new}\ m' : \mathtt{Map}[\mathtt{Agent^s\ Site}]\ \mathbf{in}\ (@_D(\mathsf{makeMap}(m'; [a_e\ s] \mathbin{::} map) \mid \\
\qquad\qquad \mathtt{lock!}m' \mid \langle a_e@s\rangle\mathtt{ack!}[])) \\
\quad \equiv @_D\langle a_e@s\rangle\mathtt{ack!}[] \mid \mathsf{mapS}_{\bullet}([a_e\ s] \mathbin{::} map)
\end{array}
$$

This implies, by Theorem 5.4.4 (Transloc Bisim Cong), $LQ_2 \dot{\succeq}^{M}_{\mathcal{E}'[\Phi]} LQ_3$ where:

$$LQ_3 \quad \equiv \quad \mathsf{mapS}_{\bullet}([a_e\ s]\texttt{::}map) \mid \textbf{new}\ \texttt{ack} : \texttt{\^{}rw}[]\ \textbf{in}\ (LR \mid @_D\langle a_e@s\rangle\texttt{ack!}[])$$

- We now need to redefine the scope of `currentloc` in order to use the temporary immobility result. For this, we define the following.

$$\begin{aligned}\mathcal{E}''[\cdot] \quad = \quad & \textbf{new}\ \Phi^{-}_{aux}, (\Delta \oplus a_e \mapsto s)\ \textbf{in} \\ & (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_{\bullet}([a_e\ s]\texttt{::}map) \mid [\cdot] \\ & \mid \textbf{new}\ \texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}\ \textbf{in}\ \mathcal{F}[\![\boldsymbol{A}']\!])\end{aligned}$$

  By Lemma D.1.7 and Lemma 6.2.7, we have $\mathcal{E}'[LQ_3] \dot{\sim}^{\emptyset}_{\Phi} \mathcal{E}''[LQ_4]$, where

$$LQ_4 \quad = \quad \textbf{new}\ \texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}, \texttt{ack} : \texttt{\^{}rw}[]\ \textbf{in}\ (LR \mid @_D\langle a_e@s\rangle\texttt{ack!}[])$$

- By (Sys-T-MigratingA), we have $\mathcal{E}'[\Phi] \vdash a_e@s$. This implies, by Lemma 6.3.5,

$$\mathcal{E}''[\Phi], \texttt{ack} : \texttt{\^{}rw}[] \Vdash @_D\langle a_e@s\rangle\texttt{ack!}[] \xrightarrow[M]{\mathsf{det}} @_{a_e}\texttt{ack!}[]$$

  By Lemma 6.4.5, we have $\textbf{new}\ \texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}\ \textbf{in}\ LR$ is temporary immobile under `ack`, wrt. $(\mathcal{E}''[\Phi], \texttt{ack} : \texttt{\^{}rw}[]), M$. Applying Lemma 6.4.2, we have: $LQ_4 \dot{\succeq}^{M}_{\mathcal{E}'[\Phi]} LQ_5$ where

$$LQ_5 \quad = \quad \textbf{new}\ \texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}, \texttt{ack} : \texttt{\^{}rw}[]\ \textbf{in}\ (LR \mid @_{a_e}\texttt{ack!}[])$$

- By Lemma 6.3.7, we have: $LQ_5 \dot{\succeq}^{M}_{\mathcal{E}'[\Phi]} LQ_6$ where:

$$\begin{aligned}LQ_6 \quad & = \quad \textbf{new}\ \texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}\ \textbf{in}\ @_{a_e}([\![P \mid Q]\!]_{a_e} \mid \texttt{Deliverer} \mid \texttt{currentloc!}s) \\ & = \quad \textbf{new}\ \texttt{currentloc} : \texttt{\^{}rw}\texttt{Site}\ \textbf{in}\ \mathcal{F}[\![[P|Q\ \mathsf{FreeA}(s)]]\!]_{a_e}\end{aligned}$$

- By Lemma D.1.3, $a_e \notin \mathsf{mayMove}(@_D \ldots, \mathcal{F}[\![\boldsymbol{A}']\!])$, this means $\mathcal{E}''[LQ_4] \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{E}''[LQ_6]$, by Theorem 5.4.3 (cong property for expansion) and transitivity of expansion.
- By Lemma D.1.7 and Lemma 6.2.7, we have $\mathcal{E}''[LQ_6] \dot{\sim}^{\emptyset}_{\Phi} \mathcal{E}'[\mathcal{F}[\![[P|Q\ \mathsf{FreeA}(s)]]\!]_{a_e}]$.

Clearly, $\mathcal{E}'[\mathcal{F}[\![[P|Q\ \mathsf{FreeA}(s)]]\!]_{a_e}] \equiv \mathcal{F}[\![Sys']\!]$ and $LP' \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{E}'[\mathcal{F}[\![[P|Q\ \mathsf{FreeA}(s)]]\!]_{a_e}]$, by transitivity of expansion. Thus the lemma is true for this case.

Case (Sys-Proc-Mesg): Supposing $Sys = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}|\mathsf{mesgReq}(\{T\}\ [a_e\ c\ v])]; \boldsymbol{A})$, Sys is idle with $\boldsymbol{A}(a_e) = [P\ \boldsymbol{E}]$. This implies, by (Sys-Proc-Mesg), $\Phi\delta \Vdash Sys \xrightarrow{\tau} Sys'$, where

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [P|c\texttt{!}v\ \boldsymbol{E}])$$

Let $\mathcal{E}[\cdot], LQ$ be defined as follows:

$$
\begin{aligned}
\mathcal{E}[\cdot] &\equiv \textbf{new } \Phi_{aux}, \Delta \textbf{ in } (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathcal{F}[\![\boldsymbol{A}]\!] \mid [\cdot]) \\
LQ &= @_D\mathsf{mesgReq}(\{T\}\,[a_e\ c\ v]) \mid \mathsf{mapS}_{\bullet}(map) \\
&= @_D(\mathsf{mesgReq}(\{T\}\,[a_e\ c\ v]) \mid \\
&\qquad \textbf{new } m : \mathtt{Map[Agent^s\ Site]} \textbf{ in } (\mathtt{lock!}m \mid \mathsf{makeMap}(m; map)))
\end{aligned}
$$

Clearly, $\mathcal{E}[LQ] \equiv \mathcal{F}[\![Sys]\!]$. In this case, we have:

- $\mathcal{E}[\Phi\delta] \Vdash LQ \xrightarrow{\tau} LQ'$, by (Lts-L-Out), (Lts-L-In) and (Lts-L-Comm), and applying (Lts-Prl) and (Lts-New), where

$$
\begin{aligned}
LQ' \equiv\ & @_D\textbf{new } m : \mathtt{Map[Agent^s\ Site]} \textbf{ in } (\mathsf{makeMap}(m; map) \\
& \mid \textbf{lookup}\mathtt{[Agent^s\ Site]}\ a_e \textbf{ in } m \textbf{ with} \\
& \quad \textbf{found}(s') \to \textbf{new } \mathtt{dack} : \mathtt{\hat{}^{rw}[]} \textbf{ in } (\langle a_e @ s'\rangle \mathtt{deliver!}\ \{T\}\,[c\ v\ \mathtt{dack}] \\
& \qquad \mid \mathtt{dack?[]} \to \mathtt{lock!}m) \\
& \quad \textbf{notfound} \to \mathbf{0})
\end{aligned}
$$

- $\Phi\delta \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow{\tau} LP'$, by (Lts-Prl) and (Lts-New), where $LP' = \mathcal{E}[LQ']$.

We now need to verify that $LP' \dot{\succeq}^{\emptyset}_{\Phi} \mathcal{F}[\![Sys']\!]$. Let $M = \mathsf{mov}(\mathcal{E}[\Phi])/\{a_e\}$.

- By (Sys-T-Map), $a_e \in \mathsf{dom}(map)$. This means there exists s_e such that

$$\mathsf{lookupL}(a_e; map) = s_e.$$

  By Lemma 6.5.3, we have $LQ' \dot{\succeq}^{M}_{\mathcal{E}[\Phi]} LQ_1$ where

$$
\begin{aligned}
LQ_1 =\ & @_D\ \textbf{new } m : \mathtt{Map[Agent^s\ Site]} \textbf{ in } (\mathsf{makeMap}(m; map) \\
& \mid \textbf{new } \mathtt{dack} : \mathtt{\hat{}^{rw}[]} \textbf{ in } (\langle a_e @ s_e\rangle \mathtt{deliver!}\ \{T\}\,[c\ v\ \mathtt{dack}] \\
& \qquad \mid \mathtt{dack?[]} \to \mathtt{lock!}m)) \\
\equiv\ & \textbf{new } \mathtt{dack} : \mathtt{\hat{}^{rw}[]} \textbf{ in } (@_D\langle a_e @ s_e\rangle \mathtt{deliver!}\ \{T\}\,[c\ v\ \mathtt{dack}] \\
& \mid @_D\textbf{new } m : \mathtt{Map[Agent^s\ Site]} \textbf{ in} \\
& \qquad (\mathsf{makeMap}(m; map) \mid \mathtt{dack?[]} \to \mathtt{lock!}m))
\end{aligned}
$$

- By Lemma D.1.4, $\mathsf{lookupL}(a_e; map) = s_e$ implies $\Phi, \Delta \vdash a_e @ s_e$. Applying Lemma 6.3.5, we have: $\mathcal{E}[\Phi] \Vdash LQ_1 \xrightarrow[M]{\mathsf{det}} LQ_2$ where

$$
\begin{aligned}
LQ_2 =\ & \textbf{new } \mathtt{dack} : \mathtt{\hat{}^{rw}[]} \textbf{ in } (@_{a_e}\mathtt{deliver!}\ \{T\}\,[c\ v\ \mathtt{dack}] \\
& \mid @_D\textbf{new } m : \mathtt{Map[Agent^s\ Site]} \textbf{ in} \\
& \qquad (\mathsf{makeMap}(m; map) \mid \mathtt{dack?[]} \to \mathtt{lock!}m))
\end{aligned}
$$

- By Lemma D.1.9, $LR = \textbf{new}\ \Omega_{aux}, \Delta\ \textbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathcal{F}[\![\boldsymbol{A}]\!])$ is temporary immobile under `lock`, wrt $(\mathcal{E}[\Phi], \Omega_D, \emptyset)$, applying Lemma 6.4.2, we have:

$$\textbf{new}\ \Omega_D\ \textbf{in}\ (LR|LQ_1) \stackrel{\cdot}{\succeq}^{\emptyset}_{\mathcal{E}[\Phi],\Delta} \textbf{new}\ \Omega_D\ \textbf{in}\ (LR|LQ_2)$$

  This implies, by Theorem 5.4.3 (cong property for expansion), $\mathcal{E}[LQ_1] \stackrel{\cdot}{\succeq}^{\emptyset}_{\mathcal{E}[\Phi]} \mathcal{E}[LQ_2]$.

- We now need to consider the reaction between the agent a_e and the daemon. Let $\boldsymbol{A}'$ be obtained by excluding a_e from the domain of $\boldsymbol{A}$ and define $\mathcal{E}'[\cdot], LR'$ as follows.

$$\begin{array}{rcl} \mathcal{E}'[\cdot] & = & \textbf{new}\ \Phi_{aux}, \Delta\ \textbf{in}\ (@_D(Daemon \mid \mathsf{mesgQ}) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \mid [\cdot]) \\ LR' & = & \mathcal{F}[\![\boldsymbol{A}(a_e)]\!]_{a_e} \\ & = & @_{a_e}([\![P]\!]_{a_e} \mid \texttt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_e} \end{array}$$

  Clearly $\mathcal{E}[LQ_2] \equiv \mathcal{E}'[LQ_2|LR']$.

- By Lemma D.1.8 and applying Lemma 6.2.6, we have $\mathcal{E}'[LQ_2|LR'] \stackrel{\cdot}{\succeq}^{M}_{\mathcal{E}[\Phi]} \mathcal{E}[LQ_3]$ where

$$\begin{array}{rcl} LQ_3 & = & \textbf{new}\ \mathsf{dack} : \texttt{\^{}rw}[]\ \textbf{in} \\ & & \quad (@_D\textbf{new}\ m : \texttt{Map[Agent}^{\texttt{s}}\ \texttt{Site]}\ \textbf{in} \\ & & \qquad (\mathsf{makeMap}(m; map) \mid \texttt{dack?}[] \rightarrow \texttt{lock!}m) \\ & & \quad \mid LR' \mid @_{a_e}(\langle D@SD \rangle \texttt{dack!}[] \mid c\texttt{!}v)) \end{array}$$

- Since D is a static agent, by Lemma 6.3.5 and Lemma 6.3.1 (Det Exp), we have

$$@_{a_e}\langle D@SD \rangle \texttt{dack!}[] \quad \succeq_{\mathcal{E}'[\Phi],\texttt{dack:\^{}rw}[]} \quad @_D\texttt{dack!}[]$$

  Applying Theorem 5.4.4 (Transloc Bisim Cong), we have $LQ_3 \succeq_{\mathcal{E}[\Phi]} LQ_4$ where:

$$\begin{array}{rcl} LQ_4 & = & \textbf{new}\ \mathsf{dack} : \texttt{\^{}rw}[]\ \textbf{in} \\ & & \quad (@_D\texttt{dack!}[] \\ & & \quad \mid @_D\textbf{new}\ m : \texttt{Map[Agent}^{\texttt{s}}\ \texttt{Site]}\ \textbf{in} \\ & & \qquad (\mathsf{makeMap}(m; map) \mid \texttt{dack?}[] \rightarrow \texttt{lock!}m) \\ & & \quad \mid LR' \mid @_{a_e}c\texttt{!}v) \end{array}$$

$$\begin{array}{rl} \equiv & \mathbf{new}\ \mathsf{dack} : \hat{\ }^{\mathtt{rw}}[]\ \mathbf{in} \\ & \quad (@_D\mathtt{dack!}[] \\ & \quad \mid @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in} \\ & \qquad (\mathsf{makeMap}(m; map) \mid \mathtt{dack?}[] \to \mathtt{lock!}m)) \\ & \mid @_{a_e}([\![c\mathtt{!}v|P]\!]_{a_e} \mid \mathtt{Deliverer}) \mid \mathcal{F}[\![\boldsymbol{E}]\!]_{a_e} \\ = & \mathbf{new}\ \mathsf{dack} : \hat{\ }^{\mathtt{rw}}[]\ \mathbf{in} \\ & \quad (@_D\mathtt{dack!}[] \\ & \quad \mid @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in} \\ & \qquad (\mathsf{makeMap}(m; map) \mid \mathtt{dack?}[] \to \mathtt{lock!}m)) \\ & \mid \mathcal{F}[\![[c\mathtt{!}v|P\ \boldsymbol{E}]]\!]_{a_e} \end{array}$$

- By Lemma 6.3.7, we have $LQ_4 \succeq_{\mathcal{E}[\Phi]} LQ_5$ where:

$$\begin{array}{rcl} LQ_5 & = & @_D\mathbf{new}\ m : \mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in}\ (\mathsf{makeMap}(m; map) \mid \mathtt{lock!}m) \\ & & \mid \mathcal{F}[\![[c\mathtt{!}v|P\ \boldsymbol{E}]]\!]_{a_e} \\ & = & \mathsf{mapS}_{\bullet}(map) \mid \mathcal{F}[\![[c\mathtt{!}v|P\ \boldsymbol{E}]]\!]_{a_e} \end{array}$$

- By Theorem 5.4.4 (Transloc Bisim Cong) and transitivity of expansion, $\mathcal{E}'[LQ_3]\dot{\succeq}^{\emptyset}_{\Phi}\mathcal{E}'[LQ_5]$.

Clearly, $\mathcal{E}'[LQ_5] \equiv \mathcal{F}[\![Sys']\!]$ and $LP'\dot{\succeq}^{\emptyset}_{\Phi}\mathcal{E}[LQ_5]$ by transitivity of expansion, thus the lemma is true for this case.

Case (Sys-Equiv): Supposing $\Phi \vdash Sys_1\ ok$, $\Phi \Vdash Sys_1 \xrightarrow[\Xi]{\beta} Sys_2$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$; moreover, $Sys_1 \equiv_{\Phi} Sys'_1$ and $Sys_2 \equiv_{\Phi,\Xi} Sys'_2$, we may derive the following. By (SYS-EQUIV), $\Phi \Vdash Sys'_1 \xrightarrow[\Xi]{\beta} Sys'_2$. By the induction hypothesis, there exists LP such that $\Phi \Vdash \mathcal{F}[\![Sys_1]\!] \xrightarrow[\Xi]{\beta} LP$ and $LP\dot{\succeq}^{\emptyset}_{\Phi,\Xi}\mathcal{F}[\![Sys_2]\!]$. Since $Sys_1 \equiv_{\Phi} Sys'_1$ implies, by Lemma D.1.2, $\mathcal{F}[\![Sys_1]\!] \equiv\overset{\alpha}{=} \mathcal{F}[\![Sys'_1]\!]$ and similarly $\mathcal{F}[\![Sys_2]\!] \equiv\overset{\alpha}{=} \mathcal{F}[\![Sys'_2]\!]$. This means that, by Theorem 4.5.2 and (LTS-CONG-R), $\Phi \Vdash \mathcal{F}[\![Sys'_1]\!] \xrightarrow[\Xi]{\beta} \mathcal{F}[\![Sys'_2]\!]$. Thus true for this case.

Therefore the lemma is proved by induction. ■

Lemma D.2.2 (Soundness of IL (7.3.4))
Given a valid system context Φ, if $\Phi \Vdash \mathcal{F}[\![Sys]\!] \xrightarrow[\Xi]{\beta} LP$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ then there exists Sys' such that $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$, and $LP\dot{\succeq}^{\emptyset}_{\Phi,\Xi}\mathcal{F}[\![Sys']\!]$.

Proof: Consider $Sys = \mathsf{eProg}(\Delta'; \boldsymbol{D}'; \boldsymbol{A}')$ well-formed wrt Φ. Since $\mathcal{F}$ is only defined if $\mathsf{dom}(\Delta') \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, we pick an injective substitution $\sigma : \mathsf{dom}(\Delta') \to \mathcal{X}/\mathsf{dom}(\Phi, \Phi_{aux})$

and denote $\Delta = \sigma\Delta'$, $\boldsymbol{D} = \sigma\boldsymbol{D}$ and $\boldsymbol{A}' = \sigma\boldsymbol{A}$. Clearly we have $Sys \stackrel{\alpha}{=} \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ and, by (L-C-Var), $\vdash \Phi, \Phi_{aux}, \Delta$. By this alpha-conversion, $\mathcal{F}[\![Sys]\!]$ is defined.

First of all we consider the flattened system according to its idle/busy status:

1. The daemon is idle: In this case, there exists no $a \in \mathsf{dom}(\boldsymbol{A})$ such that $\boldsymbol{A}(a) = [P\ \mathsf{MrdyA}(s\ P)]$ and $\mathcal{F}[\![Sys]\!]$ is as follows.

$$\begin{aligned} & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ @_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_\bullet(map) \mid \mathcal{F}[\![\boldsymbol{A}]\!] \\ = \quad & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ @_D(Daemon \mid \mathsf{mesgQ}) \\ & \quad \mid @_D\mathbf{new}\ m : \mathtt{Map[Agent^s\ Site]}\ \mathbf{in} \\ & \qquad\qquad (\mathsf{makeMap}(m; map) \mid \mathtt{lock!}m) \\ & \quad \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})}(\mathcal{F}[\![\boldsymbol{E}_a]\!]_a \mid @_a([\![P_a]\!]_a \mid \mathtt{Deliverer})) \end{aligned}$$

where, for each a, $\boldsymbol{A}(a) = [P_a\ \boldsymbol{E}_a]$.

2. The daemon is busy: In this case, there exists a unique $a_e \in \mathsf{dom}(\boldsymbol{A})$ such that $\boldsymbol{A}(a_e) = [P\ \mathsf{MrdyA}(s\ Q)]$. Letting $\boldsymbol{A}'$ be obtained by excluding a_e from the domain of $\boldsymbol{A}$, $\mathcal{F}[\![Sys]\!]$ is as follows.

$$\begin{aligned} & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ @_D(Daemon \mid \mathsf{mesgQ}) \mid \mathsf{mapS}_{a_e, \boldsymbol{A}(a_e)}(map) \mid \mathcal{F}[\![\boldsymbol{A}']\!] \\ = \quad & \mathbf{new}\ \Phi_{aux}, \Delta\ \mathbf{in}\ @_D(Daemon \mid \mathsf{mesgQ}) \\ & \quad \mathbf{new}\ \mathtt{migrated} : \mathtt{\hat{}^{rw}[Site\ \hat{}^{w}[]]}\ \mathbf{in} \\ & \qquad \mid @_D\mathbf{new}\ m : \mathtt{Map[Agent^s\ Site]}\ \mathbf{in} \\ & \qquad\qquad (\mathsf{makeMap}(m; map) \mid \mathsf{migProc}(a_e\ m\ \mathtt{migrated})) \\ & \qquad \mid @_{a_e}(\mathsf{migReady}(s\ \mathtt{migrated}\ Q) \mid [\![P]\!]_{a_e} \mid \mathtt{Deliverer}) \\ & \quad \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}}(\mathcal{F}[\![\boldsymbol{E}_a]\!]_a \mid @_a([\![P_a]\!]_a \mid \mathtt{Deliverer})) \end{aligned}$$

where, for each $a \neq a_e$, $\boldsymbol{A}(a) = [P_a\ \boldsymbol{E}_a]$.

Supposing $LP = \mathcal{F}[\![Sys]\!]$, $\Phi \vdash Sys\ ok$ and $\Phi \Vdash LP \xrightarrow[\Xi]{\beta} LQ$, by Lemma B.2.3 (Lts Analysis), the action must either be originated from one of the parallelly-composed components or their interaction. The fine-grain semantics of Nomadic π, however, prevents interaction between agents from occurring in one step. This means that the action must be originated from one of the agents (including the daemon D). The possibilities are as follows.

1. Action originated from the daemon D: Since the replicated inputs in *Daemon* and `lock` are **new**-bound at the top level, there is only one way of originating an action.
   (a) If Sys is idle then one of the message forwarding request may successfully acquires `lock`. This corresponds to the (Sys-Proc-Mesg) analysed in the proof of Lemma 7.3.3.

2. Action originated from an agent $a \in \mathsf{dom}(\boldsymbol{A})$. We first of all observe that `Deliverer` and $\mathcal{F}[\![\boldsymbol{E}]\!]_a$ cannot interact with the environment by an input or output action since the names `deliver`, `pack`, `rack`, `mack` are bound in the context. An action can be originated from an agent in the following ways.

   (a) P_a in $[\![P_a]\!]_a$ contains an **`iflocal`**, **`if`**, **`let`** or a pair of an output and a (replicated) input on the same channel. This corresponds to the rules in Figure 7.6 analysed in the proof of Lemma 7.3.3.

   (b) If $\boldsymbol{E}_a$ is $\mathsf{FreeA}(s)$ and P_a contains a **`create`** or **`migrate to`** then such a process may initialise by acquiring the `currentloc`. This corresponds to (SYS-REQ-REG) and (SYS-REQ-MIG) analysed in the proof of Lemma 7.3.3.

   Other interaction between $\mathcal{F}[\![\boldsymbol{E}_a]\!]_a$ and $[\![P_a]\!]_a$ is not possible since `pack`, `rack`, `mack` are binding in $\mathcal{F}[\![\boldsymbol{E}_a]\!]_a$. Interaction between $\mathcal{F}[\![\boldsymbol{E}_a]\!]_a$ or $[\![P_a]\!]_a$ and `Deliverer` is also not possible since `deliver` is not used for outputting in such processes.

   (c) If P_a contains a LI output $\langle b@?\rangle c!v$ then such a process may initialise a message forwarding request in the daemon. This corresponds to (SYS-REQ-MESG) analysed in the proof of Lemma 7.3.3.

   (d) If Sys is busy and a_e is ready to migrate then $\mathsf{migReady}(s\ \texttt{migrated}\ P)$ may cause a_e to migrate to s. This corresponds to (SYS-COMM-MIG) analysed in the proof of Lemma 7.3.3. Note that by ensuring that $\Phi \vdash Sys\ ok$, this migration would not be observable since a_e is **`new`**-bound within Δ.

Since all the possible transitions of $\mathcal{F}[\![Sys]\!]$ have corresponding transitions of Sys, the lemma holds.

■

D.3 Analysis of $\mathcal{D}[\![\cdot]\!]$

Lemma D.3.1 ($\mathcal{D}^{\sharp^{-1}}$ **is a progressing simulation (7.4.3))**
For any Sys, with $\Phi \vdash Sys\ ok$, if $\Phi \Vdash \mathcal{D}^{\sharp}[\![Sys]\!] \xrightarrow[\Xi]{\beta} LP$ then there exists Sys' such that $LP \equiv \mathcal{D}^{\sharp}[\![Sys']\!]$ and $\Phi \Vdash Sys \xRightarrow[\Xi]{\beta} Sys'$.

Proof: In this part of the proof, we shall validate the following diagram:

$$\begin{tikzcd}
\Phi \Vdash \mathcal{D}^\sharp [\![Sys]\!] \arrow[r, "\beta"', "\Xi"] \arrow[d, dash, "\mathcal{R}_\Phi"'] & \Phi, \Xi \Vdash \mathcal{D}^\sharp [\![Sys']\!] \arrow[d, dotted, dash, "\mathcal{R}_{\Phi,\Xi}"'] \\
\Phi \Vdash Sys \arrow[r, Rightarrow, dashed, "\beta", "\Xi"'] & \Phi, \Xi \Vdash Sys'
\end{tikzcd}$$

where the translocating relation $\mathcal{R}$ is defined as follows.

$$\mathcal{R}_\Phi \quad = \quad \{(\mathcal{D}^\sharp [\![Sys]\!], Sys) \mid \Phi \vdash Sys\ ok\}$$

Proving that $\mathcal{R}$ is a Φ_{aux}-restricted progressing simulation directly would be hard, since an action of $\mathcal{D}^\sharp [\![Sys]\!]$ may require Sys to commit some pending requests before making the corresponding action. It is sufficient, however, to prove the following

> If Sys_n with $\Phi \vdash Sys_n\ ok$ is fully-committed and $\Phi \Vdash \mathcal{D}^\sharp [\![Sys_n]\!] \xrightarrow[\Xi]{\beta} LP$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ then there exists Sys' such that $LP \equiv \mathcal{D}^\sharp [\![Sys']\!]$ and $\Phi \Vdash Sys_n \xRightarrow[\Xi]{\beta} Sys'$.

Consider Sys with $\Phi \vdash Sys\ ok$. Supposing $\Phi \Vdash \mathcal{D}^\sharp [\![Sys]\!] \xrightarrow[\Xi]{\beta} LP$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, we may derive the following. By Lemma 7.4.4, $\mathcal{D}^\sharp [\![Sys]\!] \equiv \mathcal{D}^\sharp [\![\mathsf{Norm}(Sys)]\!]$. This means that $\Phi \Vdash \mathcal{D}^\sharp [\![\mathsf{Norm}(Sys)]\!] \xrightarrow[\Xi]{\beta} LP$, by Theorem 4.5.2. By Lemma 7.4.4, $\Phi \Vdash Sys \xRightarrow{\tau} \mathsf{Norm}(Sys)$ and there exists Sys' such that $\Phi \Vdash \mathsf{Norm}(Sys) \xRightarrow[\Xi]{\beta} Sys'$ and $\mathcal{D}^\sharp [\![Sys']\!] \equiv LP$, by the above claim. This means that $\Phi \Vdash Sys \xRightarrow[\Xi]{\beta} Sys'$, by combining the transitions. This means $\mathcal{R}$ is a Φ_{aux}-restricted progressing simulation. Hence the lemma, by Lemma B.9.1.

In the rest of this proof, we shall validate the above claim. Supposing $\Phi \vdash Sys\ ok$ and $Sys = \mathsf{eProg}(\Delta''; [map''\ \mathsf{mesgQ}'']; \boldsymbol{A}'')$ is fully-committed. In order to avoid the names in Δ'' clashing with Φ_{aux} (especially when they are being extruded by output actions), we pick an injective substitution $\sigma : \mathsf{dom}(\Delta'') \to \mathcal{X}/\mathsf{dom}(\Phi, \Phi_{aux})$. We have $Sys \stackrel{\alpha}{=} \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$, where $\Delta = \sigma\Delta''$, $map = \sigma map''$, $\mathsf{mesgQ} = \sigma\mathsf{mesgQ}''$ and $\boldsymbol{A} = \sigma\boldsymbol{A}''$.

In this case, we have:

$$\begin{aligned}
\mathcal{D}^\sharp [\![Sys]\!] &= \textbf{new}\ \Delta\mathsf{commEff}(\boldsymbol{A})\ \textbf{in}\ (\mathcal{D}^\sharp [\![\mathsf{mesgQ}]\!] \mid \mathcal{D}^\sharp [\![\boldsymbol{A}]\!]) \\
&= \textbf{new}\ \Delta\ \textbf{in}\ (\mathcal{D}^\sharp [\![\mathsf{mesgQ}]\!] \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})} \mathcal{D}^\sharp [\![\boldsymbol{A}(a)]\!]_a) \\
&= \textbf{new}\ \Delta\ \textbf{in}\ ((\prod_{i \in I} @_{a_i} c_i ! v_i) \mid (\prod_{a \in \mathsf{dom}(\boldsymbol{A})} @_a P_a))
\end{aligned}$$

where $\boldsymbol{A}(a) = [P_a \ \mathsf{FreeA}(s_a)]$ for each a and $\mathsf{mesgQ} = \prod_{i\in I} \mathsf{mesgReq}\{T_i\}\,[a_i\ c_i\ v_i]$. Let $LP = \mathcal{D}^\sharp[\![Sys]\!]$ and supposing $\Phi \Vdash LP \xrightarrow[\Xi]{\beta} LQ$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, this transition can be derived from the following situations.

Case Output action from $@_{a_e} c!v$ in $\mathcal{D}^\sharp[\![\mathsf{mesgQ}]\!]$: Supposing $\Delta \equiv \Delta_1, \Delta_2$ where $\mathsf{dom}(\Delta) \cap \mathsf{fv}(v) = \mathsf{dom}(\Delta_1)$, we have, by (Lts-Open), (Lts-Prl) and (Lts-New), $\Phi \Vdash LP \xrightarrow[\Delta_1]{@_{a_e} c!v} LQ$, where

$$LQ \quad \equiv \quad \mathtt{new}\ \Delta_2\ \mathtt{in}\ (\mathcal{D}^\sharp[\![\mathsf{mesgQ}']\!] \mid \mathcal{D}^\sharp[\![\boldsymbol{A}]\!])$$

In this case, supposing $\boldsymbol{A}(a_e) = [P\ \mathsf{FreeA}(s)]$, we have:

- By (Sys-Proc-Mesg), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}']; \boldsymbol{A} \oplus a_e \mapsto [P|c!v\ \mathsf{FreeA}(s)])$$

- By (Sys-Loc-Out), we have: $\Phi \Vdash Sys' \xrightarrow[\Delta_1]{@_a c!v} Sys''$, where:

$$\begin{aligned} Sys'' \quad &= \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}']; \boldsymbol{A} \oplus a_e \mapsto [P\ \mathsf{FreeA}(s)]) \\ &= \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}']; \boldsymbol{A}) \end{aligned}$$

- $\mathcal{D}^\sharp[\![Sys'']\!] \equiv LQ$. Moreover, as clearly $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, by Lemma 7.2.1, $\Phi, \Delta_1 \vdash Sys''\ ok$. This implies $(LQ, Sys'') \in \mathcal{R}_{\Phi,\Delta_1}$.

Thus true for this case.

Case $@_{a_e} c!v$ in $\mathcal{D}^\sharp[\![\mathsf{mesgQ}]\!]$ interacts with an input in $\prod_{a\in\mathsf{dom}(\boldsymbol{A})} @_a P_a$: Similar to the previous case.

We shall now analyse possible transition of an agent in $\prod_{a\in\mathsf{dom}(\boldsymbol{A})} @_a P_a$. Supposing $a_e \in \mathsf{dom}(\boldsymbol{A})$ and $\boldsymbol{A}(a_e) = [P\ \mathsf{FreeA}(s)]$.

Case ($P \equiv c!v \mid Q$ with $\{a_e, c\} \subseteq \mathsf{dom}(\Phi)$): In this case, we have:

$$\mathcal{D}^\sharp[\![\boldsymbol{A}(a)]\!] \quad \equiv \quad @_{a_e} c!v \mid @_{a_e} Q$$

by (Lts-Open), (Lts-Prl) and (Lts-New), $\Phi \Vdash LP \xrightarrow[\Delta_1]{@_{a_e} c!v} LQ$, where

$$LQ \quad \equiv \quad \mathtt{new}\ \Delta_2\ \mathtt{in}\ (\mathcal{D}^\sharp[\![\mathsf{mesgQ}]\!] \mid @_{a_e} Q \mid \prod_{a\in\mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\sharp[\![\boldsymbol{A}(a)]\!])$$

The transition of the system can be analysed as follows:

- By (Sys-Loc-Out), we have: $\Phi \Vdash Sys \xrightarrow[\Delta_1]{@_{a_e} c!v} Sys'$, where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta_2; [map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{FreeA}(s)])$$

- $\mathcal{D}^\sharp [\![[Q\ \boldsymbol{E}]]\!]_{a_e} = @_{a_e} Q$, by the definition of $\mathcal{D}^\sharp$.
- $\mathcal{D}^\sharp [\![Sys'']\!] \equiv LQ$. Moreover, as clearly $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, by Lemma 7.2.1, $\Phi, \Delta_1 \vdash Sys''\ ok$. This implies $(LQ, Sys'') \in \mathcal{R}_{\Phi,\Delta_1}$.

Thus true for this case.

Case ($P \equiv (c?p \rightarrow R) \mid Q$ with $\{a_e, c\} \subseteq \mathsf{dom}(\Phi)$): Similar to the above case. We also omit situations where P contains **iflocal**, **if**, **let**, and a pair of output and (replicated) input on the same channel. The analysis of such cases are similar to that shown above.

Case $P \equiv (\mathbf{create}^Z\ b = P_1\ \mathbf{in}\ P_2) \mid Q$: In this case, we have:

$$\mathcal{D}^\sharp [\![\boldsymbol{A}(a_e)]\!] \quad \equiv \quad @_{a_e}(\mathbf{create}^Z\ b = P_1\ \mathbf{in}\ P_2) \mid @_{a_e} Q$$

Since (Sys-T-FreeA) implies $\Phi, \Delta \vdash a@s$, by (Lts-L-Create) we have:

$$\Phi, \Delta \Vdash_{a_e} \mathbf{create}^Z\ b = P_1\ \mathbf{in}\ P_2 \quad \xrightarrow{\tau} \quad \mathbf{new}\ b : \mathtt{Agent}^Z@s\ \mathbf{in}\ (@_{a_e} P_2 \mid @_b P_1)$$

By (Lts-Prl) and (Lts-New), we have: $\Phi \Vdash LP \xrightarrow{\tau} LQ$, where

$$\begin{aligned} LQ \quad \equiv \quad & \mathbf{new}\ \Delta\ \mathbf{in}\ (\mathcal{D}^\sharp [\![\mathsf{mesgQ}]\!] \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\sharp [\![\boldsymbol{A}(a)]\!] \\ & \quad \mid @_{a_e}(Q \mid (\mathbf{new}\ b : \mathtt{Agent}^Z@s\ \mathbf{in}\ @_{a_e} P_2 \mid @_b P_1))) \\ \equiv \quad & \mathbf{new}\ \Delta, b : \mathtt{Agent}^Z@s\ \mathbf{in}\ (\mathcal{D}^\sharp [\![\mathsf{mesgQ}]\!] \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\sharp [\![\boldsymbol{A}(a)]\!] \\ & \quad \mid @_{a_e}(Q | P_2) \mid @_b P_1) \end{aligned}$$

By (Sys-Req-Reg), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{RegA}(b\ Z\ s\ P_1\ P_2)])$$

By the definition of $\mathcal{D}^\sharp$, we have:

$$\mathcal{D}^\sharp [\![[Q\ \mathsf{RegA}(b\ Z\ s\ P_1\ P_2)]]\!]_{a_e} \quad = \quad \mathbf{new}\ b : \mathtt{Agent}^Z@s\ \mathbf{in}\ @_{a_e}(P_2 \mid Q) \mid @_b P_1$$

$\mathcal{D}^\sharp [\![Sys']\!] \equiv LQ$. Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'\ ok$, implying $(LQ, Sys') \in \mathcal{R}_\Phi$. Thus true for this case.

Case $P \equiv (\mathbf{migrate\ to}\ s \rightarrow R) \mid Q$: In this case, we have:

$$\mathcal{D}^\sharp [\![\boldsymbol{A}(a_e)]\!] \quad \equiv \quad @_{a_e}(\mathbf{migrate\ to}\ s \rightarrow R) \mid @_{a_e} Q$$

By (LTS-L-MIGRATE) we have:

$$\Phi, \Delta \Vdash_{a_e} \texttt{migrate to } s \rightarrow R \xrightarrow{@_{a_e}\mathsf{migrate\ to}\ s} @_{a_e} R$$

By (LTS-BOUND-MIG), (LTS-PRL) and (LTS-NEW), we have: $\Phi \Vdash LP \xrightarrow{\tau} LQ$, where LQ is structurally congruent to the process below.

$$\texttt{new } (\Delta \oplus a_e \mapsto s) \texttt{ in } (\mathcal{D}^\sharp[\![\mathsf{mesgQ}]\!] \mid @_{a_e}(Q|R) \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\sharp[\![\boldsymbol{A}(a)]\!])$$

By (SYS-REQ-MIG), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where:

$$Sys' = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{MtingA}(s\ R)])$$

By the definition of $\mathcal{D}^\sharp$, we have:

$$\begin{aligned} \mathcal{D}^\sharp[\![[Q\ \mathsf{MtingA}(s\ R)]]\!]_{a_e} &= @_{a_e}(Q \mid R) \\ \mathsf{commEff}(\boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{MtingA}(s\ R)]) &= a_e \mapsto s \end{aligned}$$

$\mathcal{D}^\sharp[\![Sys']\!] \equiv LQ$. Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'\ ok$, implying $(LQ, Sys') \in \mathcal{R}_\Phi$. Thus true for this case.

Case $P \equiv \langle b@?\rangle c!v \mid Q$**:** In this case, we have:

$$\mathcal{D}^\sharp[\![\boldsymbol{A}(a_e)]\!] \equiv @_{a_e}\langle b@?\rangle c!v \mid @_{a_e} Q$$

By (LTS-L-LI-SEND) we have $\Phi, \Delta \Vdash_{a_e} \langle b@?\rangle c!v \xrightarrow{\tau} @_b c!v$. By (LTS-PRL) and (LTS-NEW), we have: $\Phi \Vdash LP \xrightarrow{\tau} LQ$, where LQ is structurally congruent to the process below.

$$LQ \equiv \texttt{new } \Delta \oplus a_e \mapsto s \texttt{ in } (\mathcal{D}^\sharp[\![\mathsf{mesgQ}]\!] \mid @_{a_e} Q \mid @_b c!v \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\sharp[\![\boldsymbol{A}(a)]\!])$$

By (SYS-REQ-MESG), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$, where:

$$Sys' = \mathsf{eProg}(\Delta; [map\ \mathsf{mesgReq}(\{T\}[b\ c\ v]) \mid \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{FreeA}(s)])$$

By the definition of $\mathcal{D}^\sharp$, we have:

$$\begin{aligned} \mathcal{D}^\sharp[\![[Q\ \mathsf{FreeA}(s)]]\!]_{a_e} &= @_{a_e} Q \\ \mathcal{D}^\sharp[\![\mathsf{mesgReq}(\{T\}[b\ c\ v]) \mid \mathsf{mesgQ}]\!] &= @_b c!v \mid \mathcal{D}^\sharp[\![\mathsf{mesgQ}]\!] \end{aligned}$$

$\mathcal{D}^\sharp[\![Sys']\!] \equiv LQ$. Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'\ ok$, implying $(LQ, Sys') \in \mathcal{R}_\Phi$. Thus true for this case.

■

Lemma D.3.2 ($\mathcal{D}^\flat$ is a strict simulation (7.4.2))
For any Sys with $\Phi \vdash Sys\ ok$, if $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ then $\Phi \Vdash \mathcal{D}^\flat[\![Sys]\!] \xrightarrow[\Xi]{\hat{\beta}} \mathcal{D}^\flat[\![Sys']\!]$.

Proof: In this part of the proof, we shall try to validate the following diagram:

$$\begin{tikzcd}
\Phi \Vdash Sys \arrow[r, "\beta", "\Xi"'] \arrow[d, dash, "\mathcal{R}_\Phi"'] & \Phi, \Xi \Vdash Sys' \arrow[d, dash, dotted, "\mathcal{R}_{\Phi,\Xi}"'] \\
\Phi \Vdash \mathcal{D}^\flat[\![Sys]\!] \arrow[r, dotted, "\hat{\beta}", "\Xi"'] & \Phi, \Xi \Vdash \mathcal{D}^\flat[\![Sys']\!]
\end{tikzcd}$$

where the translocating relation $\mathcal{R}$ is defined as follows:

$$\mathcal{R}_\Phi \quad = \quad \{(\mathcal{D}^\flat[\![Sys]\!], Sys) \mid \Phi \vdash Sys\ ok\}$$

We shall prove that $\mathcal{R}^{-1}$ is a Φ_{aux}-restricted strict simulation.

Supposing $\Phi \vdash Sys\ ok$ and $Sys = \mathsf{eProg}(\Delta''; [map''\ \mathsf{mesgQ}'']; \boldsymbol{A}'')$. To avoid the names in Δ'' clashing with Φ_{aux}, we pick an injective substitution $\sigma : \mathsf{dom}(\Delta'') \to \mathcal{X}/\mathsf{dom}(\Phi, \Phi_{aux})$. We have $Sys \overset{\alpha}{=} \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}]; \boldsymbol{A})$, where $\Delta = \sigma\Delta''$, $map = \sigma map''$, $\mathsf{mesgQ} = \sigma\mathsf{mesgQ}''$ and $\boldsymbol{A} = \sigma\boldsymbol{A}''$. By the definition of $\mathcal{D}^\flat$, we have:

$$\begin{aligned}
\mathcal{D}^\flat[\![Sys]\!] &= \mathbf{new}\ \Delta\ \mathbf{in}\ \mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \mid \mathcal{D}^\flat[\![\boldsymbol{A}]\!] \\
&= \mathbf{new}\ \Delta\ \mathbf{in}\ (\prod_{i \in I} @_{a_i} c_i\,\mathbf{!}\,v_i) \mid (\prod_{a \in \mathsf{dom}(\boldsymbol{A})} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a)
\end{aligned}$$

where $\boldsymbol{A}(a) = [P_a\ \mathsf{FreeA}(s_a)]$ for each a and $\mathsf{mesgQ} = \prod_{i \in I} \mathsf{mesgReq}\{T_i\}\,[a_i\ c_i\ v_i]$. Suppose further that $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ with $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$, we shall prove the claim by an induction on the transition derivation.

Case (Sys-Loc-Tau): Suppose $\boldsymbol{A}(a_e) = [P \mid Q\ \boldsymbol{E}]$ with $P = (c\,\mathbf{!}\,v \mid c\,\mathbf{?}\,p \to R)$ and $\Phi, \Delta \Vdash_{a_e} P \xrightarrow{\tau} @_{a_e} P'$. By (SYS-LOC-TAU), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [P' \mid Q\ \boldsymbol{E}])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\mathcal{D}^\flat[\![\boldsymbol{A}(a_e)]\!]_{a_e} \quad \equiv \quad @_{a_e} P \mid @_{a_e} Q \mid \mathcal{D}^\flat[\![\boldsymbol{E}]\!]_{a_e}$$

Letting LP denote $\mathcal{D}^\flat[\![\boldsymbol{A}]\!]$, we have: By (LTS-L-OUT), (LTS-L-IN) and (LTS-L-COMM), $\Phi, \Delta \Vdash_{a_e} P \xrightarrow{\tau} @_{a_e} \mathsf{match}(p, v) R$. This implies $P' \equiv \mathsf{match}(p, v) R$. By (LTS-PRL) and (LTS-NEW), we have: $\Phi, \Delta \Vdash LP \xrightarrow{\tau} LQ$ where:

$$\begin{aligned}
LQ \quad &\equiv \quad \mathbf{new}\ \Delta\ \mathbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \mid @_{a_e}(P' \mid Q) \mid \mathcal{D}^\flat[\![\boldsymbol{E}]\!]_{a_e} \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a) \\
&= \quad \mathbf{new}\ \Delta\ \mathbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \mid \mathcal{D}^\flat[\![[P' \mid Q\ \boldsymbol{E}]]\!]_{a_e} \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a)
\end{aligned}$$

$LQ \equiv \mathcal{D}^\flat [\![Sys']\!]$. Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'\ ok$, implying $(LQ, Sys') \in \mathcal{R}_\Phi$. Thus true for this case. The case where P is an **if**, **iflocal**, **let** and a pair of output and replicated input on the same channel can be dealt with similarly.

Case (Sys-Loc-Input): Suppose $\boldsymbol{A}(a_e) = [(c?p \to P) \mid Q\ \boldsymbol{E}]$ with $a_e, c \in \mathsf{dom}(\Phi)$, Ξ is extensible, $\Phi \vdash c \in \hat{}^{\mathtt{r}}T$, $\Phi, \Xi \vdash v \in T$, $\mathsf{dom}(\Xi) \subseteq \mathsf{fv}(v)$ and $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta) = \emptyset$. We also assume further that $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$. By (Sys-Loc-Input), we have: $\Phi \Vdash Sys \xrightarrow[\Xi]{@_{a_e} c?v} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [(\mathsf{match}(p,v)P) \mid Q\ \boldsymbol{E}])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\mathcal{D}^\flat [\![\boldsymbol{A}(a_e)]\!]_{a_e} \quad \equiv \quad @_{a_e} c?p \to P \mid @_{a_e} Q \mid \mathcal{D}^\flat [\![\boldsymbol{E}]\!]_{a_e}$$

Letting LP denote $\mathcal{D}^\flat [\![\boldsymbol{A}]\!]$, we have: Since $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Delta)$, we have $\vdash \Phi, \Delta, \Xi$ and hence $\Phi, \Delta, \Xi \vdash v \in T$, by Lemma 3.7.2 (Type SW). By (Lts-L-In), $\Phi, \Delta \Vdash_{a_e} c?p \to P \xrightarrow[\Xi]{c?v} @_{a_e} \mathsf{match}(p,v)P$. By (Lts-Prl) and (Lts-New), we have: $\Phi, \Delta \Vdash LP \xrightarrow{\tau} LQ$ where:

$$\begin{aligned} LQ \quad \equiv \quad & \mathbf{new}\ \Delta\ \mathbf{in}\ (\mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \\ & \quad \mid @_{a_e}((\mathsf{match}(p,v)P) \mid Q) \mid \mathcal{D}^\flat [\![\boldsymbol{E}]\!]_{a_e} \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat [\![\boldsymbol{A}(a)]\!]_a) \\ = \quad & \mathbf{new}\ \Delta\ \mathbf{in}\ (\mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \\ & \quad \mid \mathcal{D}^\flat [\![[(\mathsf{match}(p,v)P) \mid Q\ \boldsymbol{E}]]\!]_{a_e} \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat [\![\boldsymbol{A}(a)]\!]_a) \end{aligned}$$

$LQ \equiv \mathcal{D}^\flat [\![Sys']\!]$. Moreover assuming $\mathsf{dom}(\Xi) \cap \mathsf{dom}(\Phi_{aux}) = \emptyset$ implies, by Lemma 7.2.1, $\Phi, \Xi \vdash Sys'\ ok$, that $(LQ, Sys') \in \mathcal{R}_{\Phi,\Xi}$. Thus true for this case.

Case (Sys-Loc-Output): Similar to the previous case.

Case (Sys-Req-Reg): Supposing $\boldsymbol{A}(a_e) = [P \mid Q\ \mathsf{FreeA}(s)]$ with $P = (\mathbf{create}^Z\ b = P_1\ \mathbf{in}\ P_2)$ and $b \notin \mathsf{dom}(\Phi, \Delta)$. By (Sys-Req-Reg), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{RegA}(b\ Z\ s\ P_1\ P_2)])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\mathcal{D}^\flat [\![\boldsymbol{A}(a_e)]\!]_{a_e} \quad \equiv \quad \mathbf{new}\ b : \mathtt{Agent}^Z @s\ \mathbf{in}\ @_{a_e} P \mid @_{a_e} Q$$

Letting LP denote $\mathcal{D}^\flat [\![\boldsymbol{A}]\!]$, we have:

- By (Lts-L-Create), we have: $\Phi, \Delta \Vdash_{a_e} P \xrightarrow{\tau} \mathbf{new}\ b : \mathtt{Agent}^Z @s\ \mathbf{in}\ (@_{a_e} P_2 \mid @_b P_1)$, since $\Phi, \Delta \vdash a_e @s$ (by (Sys-T-FreeA)).

- By (LTS-PRL) and (LTS-NEW), we have: $\Phi, \Delta \Vdash LP \xrightarrow{\tau} LQ$ where:

$$\begin{array}{rcl} LQ & \equiv & \textbf{new } \Delta \textbf{ in } (\mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat [\![\boldsymbol{A}(a)]\!]_a \\ & & \quad \mid @_{a_e} Q \mid \textbf{new } b : \mathtt{Agent}^Z @s \textbf{ in } (@_{a_e} P_2 \mid @_b P_1)) \\ & \equiv & \textbf{new } \Delta \textbf{ in } (\mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat [\![\boldsymbol{A}(a)]\!]_a \\ & & \quad \mid \textbf{new } b : \mathtt{Agent}^Z @s \textbf{ in } (@_{a_e} (Q \mid P_2) \mid @_b P_1)) \\ & = & \textbf{new } \Delta \textbf{in } (\mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat [\![\boldsymbol{A}(a)]\!]_a \\ & & \quad \mid \mathcal{D}^\flat [\![[Q \ \mathsf{RegA}(b \ Z \ s \ P_1 \ P_2)]]\!]_{a_e}) \end{array}$$

- $LQ \equiv \mathcal{D}^\flat [\![Sys']\!]$ Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'$ *ok*, which implies $(LQ, Sys') \in \mathcal{R}_\Phi$.

Thus true for this case.

Case (Sys-Req-Mig): Supposing $\boldsymbol{A}(a_e) = [P \mid Q \ \mathsf{FreeA}(s)]$ and $P = (\textbf{migrate to } s \rightarrow R)$. By (SYS-REQ-MIG), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [Q \ \mathsf{MtingA}(s \ R)])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\begin{array}{rcl} \mathcal{D}^\flat [\![\boldsymbol{A}(a_e)]\!]_{a_e} & = & @_{a_e}(P \mid Q) \mid \mathcal{D}^\flat [\![\mathsf{FreeA}(s)]\!]_{a_e} \\ & \equiv & @_{a_e} P \mid @_{a_e} Q \\ & = & \mathcal{D}^\flat [\![[Q \ \mathsf{MtingA}(s \ R)]]\!]_{a_e} \end{array}$$

This implies $\mathcal{D}^\flat [\![Sys']\!] = LP$. Thus true for this case.

Case (Sys-Req-Mesg): Supposing $\boldsymbol{A}(a_e) = [P \mid Q \ \boldsymbol{E}]$, $P = \langle b@?\rangle c!v$ and $\Phi, \Delta \vdash v \in T$. By (SYS-REQ-MESG), we have: $\Phi \Vdash Sys \xrightarrow{\tau} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; [\mathit{map} \ \mathsf{mesgReq}(\{T\} \ [b \ c \ v]) \mid \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [Q \ \boldsymbol{E}])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\mathcal{D}^\flat [\![\boldsymbol{A}(a_e)]\!]_{a_e} \quad \equiv \quad @_{a_e} P \mid @_{a_e} Q \mid \mathcal{D}^\flat [\![\boldsymbol{E}]\!]_{a_e}$$

Letting LP denote $\mathcal{D}^\flat [\![\boldsymbol{A}]\!]$, we have:

- By (LTS-L-LI-SEND), we have: $\Phi, \Delta \Vdash_{a_e} \ P \xrightarrow{\tau} @_b c!v$.

- By (LTS-PRL) and (LTS-NEW), we have: $\Phi, \Delta \Vdash LP \xrightarrow{\tau} LQ$ where:

$$\begin{array}{rcl} LQ & \equiv & \textbf{new}\ \Delta\ \textbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \mid (@_{a_e} Q \mid \mathcal{D}^\flat[\![\boldsymbol{E}]\!]_{a_e} \mid @_b c\,\textbf{!}\,v) \\ & & \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a) \\ & = & \textbf{new}\ \Delta\ \textbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgReq}(\{T\}\ [b\ c\ v]) \mid \mathsf{mesgQ}]\!] \mid \mathcal{D}^\flat[\![[Q\ \boldsymbol{E}]]\!]_{a_e} \\ & & \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a) \end{array}$$

- $LQ \equiv \mathcal{D}^\flat[\![Sys']\!]$ Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'\ ok$, which implies $(LQ, Sys') \in \mathcal{R}_\Phi$.

Thus true for this case.

Case (Sys-Proc-Reg): Supposing Sys is idle and $\boldsymbol{A}(a_e) = [R\ \mathsf{RegA}(b\ Z\ s\ P\ Q)]$. By (SYS-PROC-REG), we have: $\Phi \Vdash Sys \xrightarrow[\Theta]{\beta} Sys'$ where Sys' denotes the following.

$$\mathsf{eProg}(\Delta, b : \mathtt{Agent}^Z @s; [[b\ s]\,\textbf{::}\,map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [Q \mid R\ \mathsf{FreeA}(s)] \oplus b \mapsto [P\ \mathsf{FreeA}(s)])$$

We may derive the following:

$$\begin{array}{rcl} \mathcal{D}^\flat[\![Sys]\!] & = & \textbf{new}\ \Delta\ \textbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \mid \mathcal{D}^\flat[\![\boldsymbol{A}(a_e)]\!]_{a_e} \mid \displaystyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a) \\ & = & \textbf{new}\ \Delta\ \textbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a \\ & & \mid \textbf{new}\ b : \mathtt{Agent}^Z @s\ \textbf{in}\ (@_{a_e}(Q \mid R) \mid @_b P)) \\ & \equiv & \textbf{new}\ \Delta, b : \mathtt{Agent}^Z @s\ \textbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \\ & & \mid (@_{a_e}(Q \mid R) \mid @_b P) \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a) \\ & \equiv & \textbf{new}\ \Delta, b : \mathtt{Agent}^Z @s\ \textbf{in}\ (\mathcal{D}^\flat[\![\mathsf{mesgQ}]\!] \\ & & \mid \mathcal{D}^\flat[\![[Q \mid R\ \mathsf{FreeA}(s)]]\!]_{a_e} \\ & & \mid \mathcal{D}^\flat[\![[P\ \mathsf{FreeA}(s)]]\!]_b \mid \prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat[\![\boldsymbol{A}(a)]\!]_a) \\ & = & \mathcal{D}^\flat[\![Sys']\!] \end{array}$$

Thus true for this case.

Case (Sys-Proc-Mig): Supposing Sys is idle and $\boldsymbol{A}(a_e) = [Q\ \mathsf{MtingA}(s\ P)]$. By (SYS-PROC-MIG), we have: $\Phi \Vdash Sys \xrightarrow[\Theta]{\beta} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A} \oplus a_e \mapsto [Q\ \mathsf{MrdyA}(s\ P)])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\begin{array}{rcl} \mathcal{D}^\flat[\![\boldsymbol{A}(a_e)]\!]_{a_e} & = & @_{a_e} Q \mid @_{a_e}\textbf{migrate to}\ s \rightarrow P \\ & = & \mathcal{D}^\flat[\![[Q\ \mathsf{MrdyA}(s\ P)]]\!]_{a_e} \end{array}$$

This implies $\mathcal{D}^\flat [\![Sys']\!] = LP$. Thus true for this case.

Case (Sys-Comm-Mig): Supposing $\boldsymbol{A}(a_e) = [Q\ \mathsf{MrdyA}(s\ P)]$. By (Sys-Proc-Mig-Comm), we have: $\Phi \Vdash Sys \xrightarrow[\Theta]{\beta} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta @_{a_e} \mathsf{migrate\ to}\ s; [[a_e\ s] \mathbf{::} map\ \mathsf{mesgQ}]; \boldsymbol{A} \oplus a_e \mapsto [P \mid Q\ \mathsf{FreeA}(s)])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\mathcal{D}^\flat [\![\boldsymbol{A}(a_e)]\!]_{a_e} \quad = \quad @_{a_e} Q \mid @_{a_e} \mathtt{migrate\ to}\ s \rightarrow P$$

Letting LP denote $\mathcal{D}^\flat [\![\boldsymbol{A}]\!]$, we have:

- By (Lts-L-Mig), we have: $\Phi, \Delta \Vdash_{a_e} \mathtt{migrate\ to}\ s \rightarrow P \xrightarrow{\mathsf{migrate\ to}\ s} @_{a_e} P$.
- By (Lts-Bound-Mig), (Lts-Prl) and (Lts-New), we have: $\Phi, \Delta \Vdash LP \xrightarrow{\tau} LQ$ where:

$$\begin{aligned} LQ \quad &\equiv \quad \mathtt{new}\ (\Delta \oplus a_e \mapsto s)\ \mathtt{in}\ (\mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \\ &\qquad \mid (@_a Q \mid @_a P) \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat [\![\boldsymbol{A}(a)]\!]_a) \\ &= \quad \mathtt{new}\ (\Delta \oplus a_e \mapsto s)\ \mathtt{in}\ (\mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \\ &\qquad \mid \mathcal{D}^\flat [\![[P \mid Q\ \mathsf{FreeA}(s)]]\!]_{a_e} \mid \textstyle\prod_{a \in \mathsf{dom}(\boldsymbol{A})/\{a_e\}} \mathcal{D}^\flat [\![\boldsymbol{A}(a)]\!]_a) \end{aligned}$$

- $LQ \equiv \mathcal{D}^\flat [\![Sys']\!]$. Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'\ ok$, which implies $(LQ, Sys') \in \mathcal{R}_\Phi$.

Thus true for this case.

Case (Sys-Proc-Mesg): Supposing Sys is idle and $\mathsf{mesgQ} \equiv \mathsf{mesgReq}(\{T\}\ [a_e\ c\ v]) \mid \mathsf{mesgQ}'$ with $\boldsymbol{A}(a_e) = [P\ \boldsymbol{E}]$. By (Sys-Proc-MesgQ), we have: $\Phi \Vdash Sys \xrightarrow[\Theta]{\beta} Sys'$ where:

$$Sys' \quad = \quad \mathsf{eProg}(\Delta; [map\ \mathsf{mesgQ}']; \boldsymbol{A} \oplus a_e \mapsto [P \mid c\mathbf{!}v\ \boldsymbol{E}])$$

By the definition of $\mathcal{D}^\flat$, we have:

$$\begin{aligned} \mathcal{D}^\flat [\![\mathsf{mesgQ}]\!] \mid \mathcal{D}^\flat [\![\boldsymbol{A}(a_e)]\!]_{a_e} \quad &= \quad @_{a_e} c\mathbf{!}v \mid \mathcal{D}^\flat [\![\mathsf{mesgQ}']\!] \mid @_{a_e} P \mid \mathcal{D}^\flat [\![\boldsymbol{E}]\!]_{a_e} \\ &= \quad \mathcal{D}^\flat [\![\mathsf{mesgQ}']\!] \mid \mathcal{D}^\flat [\![[P \mid c\mathbf{!}v\ \boldsymbol{E}]]\!]_{a_e} \end{aligned}$$

This implies $\mathcal{D}^\flat [\![Sys']\!] = LP$. Moreover, by Lemma 7.2.1, $\Phi \vdash Sys'\ ok$, which implies $(LP, Sys') \in \mathcal{R}_\Phi$. Thus true for this case.

■

Bibliography

[AB96] A. Asperti and N. Busi. Mobile Petri Nets. Technical report, Laboratory for Computer Science, University of Bologna, Italy, 1996.

[ABL99] Roberto M. Amadio, Gérard Boudol, and Cedric Lhoussaine. The receptive distributed pi-calculus. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Proceedings of FSTTCS '99*, volume 1738 of *LNCS*. Springer, December 1999.

[ACM96] ACM. *23rd Annual Symposium on Principles of Programming Languages (POPL) (St. Petersburg Beach, Florida)*, January 1996.

[ACM97] ACM. *24th Annual Symposium on Principles of Programming Languages (POPL) (Paris, France)*, January 1997.

[ACM00] ACM. *27th Annual Symposium on Principles of Programming Languages (POPL) (Boston, Mass.)*, 2000.

[ACM01] ACM. *28th Annual Symposium on Principles of Programming Languages (POPL) (London, England)*, 2001.

[ACS98] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.

[AD94] Roberto Amadio and Mads Dam. Reasoning about higher-order processes. Technical Report R94-18, Swedish Institute of Computer Science, November 20, 1994.

[AF89] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9), September 1989.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01* [ACM01], pages 104–115.

[AFG98] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of LICS '98* [IEE98], pages 105–116.

[AFG00] Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of POPL '00* [ACM00], pages 302–315.

[AG97] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In Mazurkiewicz and Winkowski [MW97], pages 59–73.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 143:1–70, 1999. An extended abstract appeared in the *Proceedings of the Fourth ACM Conference on Computer and Communications Security (Zürich, April 1997)*. An extended version of this paper appears as Research Report 149, Digital Equipment Corporation Systems Research Center, January 1998, and, in preliminary form, as Technical Report 414, University of Cambridge Computer Laboratory, January 1997.

[Aït95] Otmane Aït-Mohamed. Mechanizing a pi-calculus equivalence in HOL. In *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971, Aspen Grove, Utah, September 1995.

[AKH90] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. Technical Report 5/90, Computer Science, School of Cognitive and Computing Sciences, University of Sussex, Brighton BN1 9QH, July 1990.

[ALT95] Roberto M. Amadio, Lone Leth, and Bent Thomsen. From a concurrent λ-calculus to the π-calculus. In Horst Reichel, editor, *Proceedings of FCT '95*, volume 965 of *Lecture Notes in Computer Science*, pages 106–115. Springer, 1995. Full version as Technical Report ECRC-95-18.

[Ama94] Roberto M. Amadio. Translating Core Facile. Technical Report ECRC-94-3, European Computer-Industry Research Centre, München, February 1994.

[Ama97] Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In D. Garlan and D. Le Metayer, editors, *Proceedings of COORDINA-*

TION '97, volume 1282 of *Lecture Notes in Computer Science*. Springer, 1997. Extended version as Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

[AP94] Roberto M. Amadio and Sanjiva Prasad. Localities and failures. In Pazhamaneri S. Thiagarajan, editor, *Proceedings of FSTTCS '94*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 1994. Full version as Technical Report 94-18, ECRC Munich, 1994.

[AP98] Roberto M. Amadio and Sanjiva Prasad. Modelling IP mobility. In Davide Sangiorgi and Robert de Simone, editors, *Proceedings of CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*, pages 301–316. Springer, September 1998.

[ARS97] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: a language for resource aware mobile programs. In *Mobile Object Systems*, number 1222 in Lecture Notes in Computer Science, pages 111–130. Springer Verlag (D), February 1997.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BFL98] Michele Boreale, Cédric Fournet, and Cosimo Laneve. Bisimulations for the join-calculus. In David Gries and Willem-Paul de Roever, editors, *Proceedings of PROCOMET '98*. IFIP, Chapman & Hall, 1998.

[BGL94] A. Bouali, S. Gnesi, and S. Larosa. The integration project for the JACK environment. *Bullettin of the EATCS*, 54, 1994.

[BHDH98] Michael Bursell, Richard Hayton, Douglas Donaldson, and Andrew Herbert. A mobile object workbench. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents (MA)*, volume 1477 of *Lecture Notes in Computer Science*, pages 136–147, Berlin, Heidelberg, New York, Tokyo, September 1998. Springer-Verlag.

[BL85] A. Barak and A. Litman. MOS — a multicomputer distributed operating system. *Practice & Experience*, 15(8):725–737, August 1985.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *ACM Principles of Programming Languages*, January 1992.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[Bou94] Gérard Boudol. Some chemical abstract machines. In Jacobus W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *A Decade of Concurrency 1993 (REX Workshop)*, volume 803 of *Lecture Notes in Computer Science*, pages 92–123. Springer, 1994.

[BS85] A. Barak and A. Shiloh. A distributed load balancing policy for a multicomputer. *Practice & Experience*, 15(9), September 1985.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. Also available as Digital Systems Research Center Research Report 122.

[Car99] L. Cardelli. Abstractions for mobile computations. *Lecture Notes in Computer Science*, 1603:51–94, 1999.

[CF99] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages 22–29, Palm Springs, CA, USA, October 1999.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Proceedings of FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[CG00] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of POPL '00* [ACM00].

[CG01] Luca Cardelli and Andrew D. Gordon. Logical properties of name restriction. In *the proceedings of Foundations of Software Science and Computation Structures (FoSSaCS'01)*, Lecture Notes in Computer Science, 2001. To appear.

[CGG99] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiří Wiederman, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings of ICALP '99*, volume 1644 of *Lecture Notes in Computer Science*, pages 230–239. Springer, July 1999.

[CGG00] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of*

Theoretical Informatics, Proceedings of the International IFIP Conference TCS 2000 (Sendai, Japan), volume 1872 of *LNCS*, pages 333–347. IFIP, Springer, August 2000.

[CGZ95] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems*, volume 924, pages 66–76. Springer Verlag, 1995.

[CHK97] D. Chess, C. G. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, Lecture Notes in Computer Science, pages 25–47. Springer-Verlag, Berlin Germany, 1997.

[Cho97] G. Chockler. An adaptive totally ordered multicast protocol that tolerates partitions. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, August 1997.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design : a Foundation.* Addison-Wesley, Reading, Mass., 1988.

[CP99] Mozart Consortium and Mozart Programming. *The Mozart Programming System.* Programming Systems Lab, Saarbrucken and Swedish Institute of Computer Science and Stockholm and Universit'e catholique de Louvain, 1999. Documentation and system available from http://www.mozart-oz.org.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Proceedings of CAV'96 (Computer Aided Verification)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, NJ, USA, 1996. Springer-Verlag.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CV99] Giuseppe Castagna and Jan Vitek. Commitment and confinement for the seal calculus. Trusted objects, Centre Universitaire d'Informatique, University of Geneva, July 1999.

[Dam95] Mads Dam. Compositional proof systems for model checking infinite state processes. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, pages 12–26, Philadelphia, Pennsylvania, 21–24 August 1995. Springer-Verlag.

[Dam01] Mads Dam. Proof systems for pi-calculus logics. In R. de Queiroz, editor, *Logic for Concurrency and Synchronisation*, Studies in Logic and Computation. Oxford University Press, 2001. To appear.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Technical Report 154, INRIA, Rocquencourt, France, 1993.

[dH84] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes (concurrent programming). *Theoretical Computer Science*, 34(1-2):83–133, November 1984.

[dNFP97a] R. de Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. Technical report, Dipartimento di Sistemi e Informatica, Universit'a di Firenze, Florence, 1997.

[dNFP97b] R. de Nicola, G. Ferrari, and R. Pugliese. Locality based Linda: programming with explicit localities. In *Proceedings of FASE–TAPSOFT'97*. Springer-Verlag, 1997.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software: Practice and Experience*, 21(8):767–785, 1991.

[EE98] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[Fes98] Fabrice Le Fessant. *The JoCaml reference manual*. INRIA Rocquencourt, 1998. Available from http://join.inria.fr.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96* [ACM96], pages 372–385.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Montanari and Sassone [MS96], pages 406–421.

[FHJ98] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. *Journal of Functional Programming*, 8(5):447–491, 1998.

[FKJ93] F. Andersen, K.D. Petersen, and J.S. Petterson. Program Verification using HOL-UNITY. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 1–16, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[FKL95] A. Fekete, M. F. Kaashoek, and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 439–449, Los Alamitos, CA, USA, May 30–June 2 1995. IEEE Computer Society Press.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), 1985.

[FLS98] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *16th Annual ACM Symposium on Principles of Distributed Computing*, August 1998.

[FM97] Cédric Fournet and Luc Maranget. *The join-calculus language (version 1.03)*, 1997. Source distribution and documentation available from http://join.inria.fr/.

[Fou98] Cédric Fournet. *The Join-Calculus: A Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, Paris, France, 1998.

[FPV98] A. Fuggetta, G. Picco, and G. Vigna. Understanding code mobility. *IEEE Trans. on Software Engineering*, 24(5):342–361, May 1998.

[FZ94] G. Forman and J. Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(6), April 1994.

[GC99] Andrew D. Gordon and Luca Cardelli. Equational Properties of Mobile Ambients. In Wolfgang Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 1999.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1997.

[GL98] S. Garland and N. Lynch. The IOA language and toolset: Support for designing. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998.

[Gla98] G. Glass. Objectspace voyager core package technical overview. Technical report, Objectspace, 1998.

[GLV97] S. J. Garland, N. A. Lynch, and M. Vaziri. IOA: A language for specifiying, programming, and validating distributed systems, September 1997. Available through URL `http://larch.lcs.mit.edu:8001/~garland/ioaLanguage.html`.

[Gor87] M. Gordon. HOL: A proof generating system for higher-order logic. In Birtwistle and Subrahmanyam, editors, *VSLI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1987.

[Gra96] Robert S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, Monterey, Cal., July 1996.

[Gro] Internet Engineering Task Force MANET Working Group. Mobile ad-hoc networks (manet) charter. Available at http://www.ietf.org/html.charters/manet-charter.html.

[GS95] J. F. Groote and M. P. A. Sellink. Confluence for process verification. In Lee and Smolka [LS95], pages 204–218.

[HC96] B. Heyd and P. Crégut. A modular coding of UNITY in Coq. In J. von Wright, T. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 251– 266, 1996.

[Hen88] Matthew Hennessey. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hir97] Daniel Hirschkoff. A full formalization of pi-calculus theory in the Calculus of Constructions. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher-Order Logic: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 153–169, 1997.

[Hir98] Daniel Hirschkoff. Automatically proving up to bisimulation. In Petr Jancar and Mojmir Kretinsky, editors, *Proceedings of MFCS '98 Workshop on Concurrency*, volume 18 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science Publishers, 1998.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 1985.

[Hoa69] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HP94] G. J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[HT91] Kohei Honda and Mario Tokoro. A small calculus for concurrent objects. *ACM OOPS Messenger*, 2(2):50–54, 1991.

[IEE98] IEEE. *Thirteenth Annual Symposium on Logic in Computer Science (LICS) (Indiana)*. Computer Society Press, July 1998.

[IJ93] J. Ioannidis and G. Q. Maguire Jr. The design and implementation of mobile internetworking architecture. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 491–502, January 1993.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of POPL '01* [ACM01], pages 128–141.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JNW99] Daniel Jackson, Yu-Chung Ng, and Jeannette M. Wing. A Nitpick Analysis of Mobile IPv6. *Formal Aspects of Computing*, 11(6):591–615, 1999.

[JR00] A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. In *the 15th Annual IEEE Symposium on Logic in Computer Science, LICS'00*, Santa Barbara, 2000.

[JRS94] Dag Johansen, Robbert Van Renesse, and Fred B. Schneider. Operating system support for mobile agents. Technical Report TR94-1468, Cornell University, Computer Science Department, 1994.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of POPL '96* [ACM96], pages 358–371.

[LAFSV00] Luís Lopes, Álvaro Figueira, Fernando Silva, and Vasco Vasconcelos. A concurrent programming environment with support for distributed computations and code mobility. Technical Report DCC-2000-5, DCC - FC & LIACC, Universidade do Porto, June 2000. (extended version of the paper presented at Cluster'2000).

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Engin*, 3:125–143, 1977.

[LDG+00] X. Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System: Documentation and User's Manual*, 2000. Available from http://caml.inria.fr/.

[Lin94] Huimin Lin. Symbolic bisimulation and proof systems for the π-calculus. Technical Report 7/94, School of Cognitive and Computing Sciences, University of Sussex, UK, 1994.

[LO98] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[LS92] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In *Proc. of the Winter 1992 USENIX*, pages 283–290, January 1992.

[LS95] Insup Lee and Scott A. Smolka, editors. *CONCUR '95: Concurrency Theory, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, Philadelphia, Pennsylvania, 21–24August 1995. Springer-Verlag.

[LSFV99] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: An experiment in code mobility from the realm of process calculi. In *5 th Mobile Object Systems Workshop*, 1999. part of ECOOP'99.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology, April 1987.

[LT88] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1988.

[LV94] N. Lynch and F. Vaandrager. Forward and backward simulations part I: Untimed systems (replaces TM-486). Technical Memo MIT/LCS/TM-486b, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1994.

[LW95] Xinxin Liu and David Walker. Confluence of processes and systems of objects. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwarzbach, editors, *Proceedings of TAPSOFT '95*, volume 915 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 1995. Presented in the CAAP-section. Available as University of Warwick Research Report CS-RR-272, October 1994.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, USA, 1997.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1996.

[Mas99] C. Mascolo. Mobis: A specification language for mobile systems. In P. Ciancarini and A. Wolf, editors, *Proceedings of Coordination Languages and Models, Coordination'99*, volume 1594 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, 1999.

[MCR+96] D. Milojičić, M. Condict, F. Reynolds, D. Bolinger, and P. Dale. Mobile objects and agents. In *Distributed Object Computing on the Internet AdvancedTopics*

Workshop, Second USENIX Conference on Object Oriented Technologies and Systems (COOTS), Toronto, Canada, 1996.

[MDW99] Dejan Milojičić, Frederick Douglis, and Richard Wheeler, editors. *Mobility: processes, computers, and agents.* Addison-Wesley, Reading, MA, USA, 1999.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992. Previous version as Rapport de Recherche 1154, INRIA Sophia-Antipolis, 1990, and in *Proceedings of ICALP '91*, LNCS 443.

[Mil93a] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1):78–89, 1993. Turing Award Lecture.

[Mil93b] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F: Computer and System Sciences.* NATO Advanced Study Institute, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[MMvR95] M. Muller, T. Muller, and P. van Roy. Multiparadigm programming in Oz. In *Workshop on the Future of Logic Programming, International Logic Programming Symposium (ILSP 95)*, December 1995.

[Mor99] Luc Moreau. Distributed Directory Service and Message Router for Mobile Agents. Technical Report ECSTR M99/3, University of Southampton, ECSTR M99/3, 1999.

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.

[MPW93] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.

[MR99] Peter J. McCann and Gruia-Catalin Roman. Modeling mobile IP in mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, 8(2):115–146, April 1999.

[MS] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench User Manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.

[MS96] Ugo Montanari and Vladimiro Sassone, editors. *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, Pisa, Italy, 26–29August 1996. Springer-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[MW97] Antoni Mazurkiewicz and Józef Winkowski, editors. *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, Warsaw, Poland, July 1997. Springer-Verlag.

[Nec97] G. Necula. Proof-carrying code. In *Proceedings of POPL '97* [ACM97].

[Nes92] M. Nesi. A formalization of the process algebra CCS in higher order logic. Technical Report 278, University of Cambridge, Computer Laboratory, December 1992.

[Nes96] Uwe Nestmann. *On Determinacy and Nondeterminacy in Concurrent Programming*. PhD thesis, Technische Fakultät, Universität Erlangen, November 1996. Arbeitsbericht IMMD-29(14).

[Nes97] Uwe Nestmann. What is a 'good' encoding of guarded choice? In Catuscia Palamidessi and Joachim Parrow, editors, *Proceedings of EXPRESS '97*, volume 7 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science Publishers, 1997. Full version as report BRICS-RS-97-45, Universities of Aalborg and Århus, Denmark, 1997. Revised version accepted (1998) for *Journal of Information and Computation*.

[Nes98] Uwe Nestmann. On the expressive power of joint input. In Catuscia Palamidessi and Ilaria Castellani, editors, *Proceedings of EXPRESS '98*, volume 16.2 of

Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier Science Publishers, 1998.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Montanari and Sassone [MS96], pages 179–194. Revised full version as report ERCIM-10/97-R051, European Research Consortium for Informatics and Mathematics, 1997.

[NR00] U. Nestmann and A. Ravara. What's TyCO, after all? Research report, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, 1049-001 Lisboa, Portugal, 2000. Short abstract. Presented at Express'00, satellite event of CONCUR'00.

[OCD+87] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch. The Sprite Network Operating System. Technical Report UCB/CSD/87/359, University of California, Berkeley, California 94720, 1987.

[OMG96] OMG. The Common Object Request Broker Architecture: Architecture and specification. Technical Report PTC/96-03-04, Object Management Group, 1996.

[Pal97] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of POPL '97* [ACM97], pages 256–265.

[Par81] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, Heidelberg, and New York, March 1981. Springer-Verlag.

[Par99] Joachim Parrow. Trios in concert. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. Massachusetts Institute of Technology, 1999.

[Par00] Joachim Parrow. An introduction to the π-calculus, 2000. Chapter to appear in Bergstra, Ponse and Smolka (Eds) Process Algebra, Elsevier 2000.

[Pau99] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. Technical Report 467, Computer Laboratory, University of Cambridge, May 1999.

[Phi96] Anna Philippou. *Reasoning about Systems with Evolving Structure*. PhD thesis, University of Warwick, December 1996.

[PMR99] G. Picco, A. Murphy, and G. Roman. Lime: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21 st International Conference on Software Engineering*, May 1999.

[PRM97] Gian Pietro Picco, Gruia-Catalin Roman, and Peter J. McCann. Reasoning about Code Mobility with Mobile UNITY. Technical Report WUCS-97-43, Washington University, St. Louis, MO, USA, December 1997.

[PS92] J. Parrow and P. Sjodin. Multiway synchronization verified with coupled simulation. *Lecture Notes in Computer Science*, 630:518–533, 1992.

[PS94a] J. Parrow and P. Sjodin. The complete axiomatization of cs-congruence. *Lecture Notes in Computer Science*, 775:557–568, 1994.

[PS94b] Benjamin Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extract appeared in *Proceedings of LICS '93*: 376–385.

[PS97] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Proceedings of POPL '97* [ACM97], pages 241–255. Full version available as University of Pennsylvania Technical Report MS-CIS-99-10. Previous version as INRIA-Sophia Antipolis Rapport de Recherche No. 3042 and as Indiana University Computer Science Technical Report 468.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[PV98] Joachim Parrow and Björn Victor. The Fusion Calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS '98* [IEE98], pages 176–185.

[Qin91] Huajin Qin. Efficient verification of determinate processes. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR '91*, volume 527 of *Lecture Notes in Computer Science*, pages 470–479, Amsterdam, The Netherlands, 26–29August 1991. Springer-Verlag.

[Rei85] Wolfgang Reisig. Petri nets. An Introduction. In W. Brauer, G. Rozenberg, and A. Salomaa, editors, *EATCS Monographs on Theoretical Compute Science*, volume 4. Springer-Verlag, Berlin, Germany, 1985.

[Rep92] John H. Reppy. *Higher-order concurrency.* PhD thesis, Cornell University, Ithaca, NY, 1992. Technical Report TR 92-1285.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Proceedings of ICALP '97*, volume 1256 of *Lecture Notes in Computer Science*, pages 471–481. Springer, 1997. Full version as CogSci Report 2/97, University of Sussex, Brighton.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL '98*. ACM, 1998.

[RH99] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL '99*, pages 93–104. ACM, 1999. Full version as CogSci Report 4/98, University of Sussex, Brighton.

[RMP97] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.

[San93a] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, LFCS, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).

[San93b] Davide Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *Proceedings of TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1993.

[San94a] Davide Sangiorgi. An investigation into functions as processes. In Steve Brookes, Michael Main, Austin Melton, and David Schmidt, editors, *Proceedings of MFPS '93*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 1994.

[San94b] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Journal of Information and Computation*, 111(1):120–131, 1994. An extended abstract appeared in *Proceedings of LICS '92*.

[San95a] Davide Sangiorgi. Lazy functions and mobile processes. Rapport de Recherche RR-2515, INRIA Sophia-Antipolis, 1995.

[San95b] Davide Sangiorgi. On the proof method for bisimulation (extended abstract). In J. Wiedemann and P. Hájek, editors, *Proceedings of MFCS '95*, volume 969 of *Lecture Notes in Computer Science*, pages 479–488. Springer, 1995. To appear in *Mathematical Structures in Computer Science*. Full version available electronically.

[San96a] Davide Sangiorgi. Bisimulation in higher-order process calculi. *Journal of Information and Computation*, 131:141–178, 1996. Available as Rapport de Recherche RR-2508, INRIA Sophia-Antipolis, 1995. An early version appeared in *Proceedings of PROCOMET'94*, pages 207–224. IFIP. North Holland Publisher.

[San96b] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh. An extended abstract appeared in the *Proceedings of CONCUR '93*, LNCS 715.

[San99] Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999. An abstract appeared in the *Proceedings of ICALP '97*, LNCS 1256, pages 303–313.

[Sew97] Peter Sewell. On implementations and semantics of a concurrent programming language. In Mazurkiewicz and Winkowski [MW97], pages 391–405.

[Sew98] Peter Sewell. Global/local subtyping and capability inference for a distributed pi-calculus. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *Lecture Notes in Computer Science*, pages 695–706. Springer, July 1998. Full version as Technical Report 435, Computer Laboratory, University of Cambridge.

[Sew00] Peter Sewell. A brief introduction to applied π. Technical Report 498, Computer Laboratory, University of Cambridge, Cambridge, UK, August 2000.

[SM92] D. Sangiorgi and R. Milner. The problem of "weak bisimulation up to". In W. R. Cleaveland, editor, *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, Stony Brook, New York, 24–27August 1992. Springer-Verlag.

[Smi97] Mark Smith. *Formal Verification of TCP and T/TCP.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, September 1997.

[Smo94] Gert Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *Proceedings 1st International Conference of Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72. Springer, 1994. Available as Research Report RR-94-16 from DFKI Kaiserslautern.

[Smo95a] G. Smolka. The definition of kernel Oz. In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. SpringerVerlag, Berlin, 1995.

[Smo95b] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, pages 324–343. Springer-Verlag, Berlin, 1995.

[SV99] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *Proceedings of CSFW 99: The 12th IEEE Computer Security Foundations Workshop (Mordano, Italy)*, pages 136–150. IEEE Computer Society, June 1999.

[SV00] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Proceedings of CSFW 00: The 13th IEEE Computer Security Foundations Workshop.*, pages 269–284. IEEE Computer Society, July 2000. Full versions with title *Secure Composition of Untrusted Code: Wrappers and Causality Types* appears as Technical Report 478, Computer Laboratory, University of Cambridge, 1999.

[SWP99] Peter Sewell, Pawel Wojciechowski, and Benjamin Pierce. Location independence for mobile agents. In H.E. Bal, B. Belkhouche, and Luca Cardelli, editors, *Proceedings of ICCL '98, Workshop on Internet Programming Languages (Chicago, IL, USA, May 13, 1998)*, volume 1686 of *Lecture Notes in Computer Science*. Springer, September 1999. Full version with title *Location-Independent*

Communication for Mobile Agents: a Two-Level Architecture appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

[SY97] Tatsurou Sekiguchi and Akinori Yonezawa. A calculus with code mobility. In *Prodeedings of FMOODS'97, Canterbury, July.* IFIP, Chapman and Hall, 1997.

[TLG92] B. Thomsen, L. Leth, and A. Giacalone. Some issues in the semantics of Facile distributed programming. In *Proceedings of REX Workshop Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*. Springer, 1992.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In Montanari and Sassone [MS96], pages 278–298.

[TLP+93] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz C. Knabe, and Alessandro Giacalone. Facile antigua release programming guide. Technical Report ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, December 1993.

[Tof91] Chris Tofts. *Proof Methods and Pragmatics for Parallel Programming.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, February 1991. Published as ECS-LFCS-91-140.

[Tsc94] C. F. Tschudin. An Introduction to the MØ Messenger Language. Technical Report Cahier du CUI No 86, University of Geneva, Switzerland, 1994.

[TSS+97] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications*, pages 80–86, January 1997.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, LFCS, University of Edinburgh, June 1996. CST-126-96 (also published as ECS-LFCS-96-345).

[US01] Asis Unyapoth and Peter Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL '01* [ACM01], pages 116–127.

[Vas94] Vasco Thudichum Vasconcelos. *A process-calculus approach to typed concurrent objects.* PhD thesis, Keio University, 1994.

[VC98] Jan Vitek and Giuseppe Castagna. Towards a calculus of secure mobile computations. In *IEEE Workshop on Internet Programming Languages*, Chicago, Illinois, May 1998.

[VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *In Internet Programming Languages*, 1999.

[vG90] R. J. van Glabbeek. The linear time – branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *Proceedings of CONCUR'90*, LNCS 458, pages 278–297. Springer-Verlag, 1990.

[vG93] R. J. van Glabbeek. The linear time – branching time spectrum II (the semantics of sequential systems with silent moves). In *Proceedings of CONCUR '93*, pages 66–81, 1993.

[Vic94] Björn Victor. *A Verification Tool for the Polyadic π-Calculus.* Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[Vic98] Björn Victor. *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes.* PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden, June 1998.

[VLS98] Vasco Thudicum Vasconcelos, Luìs Lopes, and Fernando Silva. Distribution and mobility with lexical scoping in process calculi. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science Publishers, 1998.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π-calculus. In David Dill, editor, *Proceedings of CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer, 1994.

[vO94] Vincent van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126(2):259–280, April 1994.

[VP96] Björn Victor and Joachim Parrow. Constraints as processes. In Montanari and Sassone [MS96], pages 389–405.

[VvRB98] W. Vogels, R. van Renesse, and K. Birman. Six misconceptions about reliable distributed computing. In *Proceedings of 5 th SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[Wal95] David Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communication of the ACM*, 36(7):74–84, July 1993.

[Whi95] J. White. Telescript technology: The foundation for the electronic marketplace. Technical report, General Magic, Inc, Sunnyvale, CA, USA, 1995. Whitepaper.

[WMG00] B. Wang, J. Meseguer, and C. Gunter. Specification and formal verification of a PLAN algorithm in Maude. In *Proceedings of the International workshop on Distributed System Valdiation and Verification*, pages 49–56, 2000.

[WMLF98] P. Wycko, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, August 1998.

[Woj00a] Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, Computer Laboratory, University of Cambridge, 2000. Available as Technical Report 492, June 2000.

[Woj00b] Paweł T. Wojciechowski. *The Nomadic Pict System Release 1.0a.* Computer Laboratory, University of Cambridge, December 2000. Available from http://www.cl.cam.ac.uk/users/ptw20/nomadicpict.html.

[WPW98] T. Walsh, N. Paciorek, and D. Wong. Security and reliability in Concordia. In *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences*, volume VII, pages 44–53, January 1998.

[WS99] Pawel Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. In *Proceedings of ASA/MA '99 (First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents)*, October 1999. An extended version appeared in IEEE Concurrency vol 8 no 2, 2000.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc, 1994.

[YH99a] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. CogSci Report 99.02, School of Cognitive and Computing Sciences, University of Sussex, UK, 1999.

[YH99b] Nobuko Yoshida and Matthew Hennessy. Subtyping and locality in distributed higher order processes. In Jos C.M. Baeten and Sjouke Mauw, editors, *Proceedings of CONCUR '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 557–572. Springer, August 1999.

Technical Terms