

Daniele Gorla

Semantic Approaches to Global Computing Systems

PhD Thesis

Dottorato in Informatica ed Applicazioni, ciclo XVII

Dipartimento di Sistemi ed Informatica

Università degli Studi di Firenze

Advisors:

Prof. Rocco De Nicola

Prof. Rosario Pugliese

International Reviewers:

Dr. Gérard Boudol

Prof. Michele Bugliesi

Members of the Jury:

Prof. Pierluigi Crescenzi

Prof. Roberto Grossi

Prof. Davide Sangiorgi

Date of submission: December 31, 2004

Date of defence: February 15, 2005

Abstract

Programming computational infrastructures available globally for offering uniform services has become one of the main issues in Computer Science. The challenges come from the variable guarantees for communication, co-operation and mobility, resource usage, security policies and mechanisms, etc. that have to be taken into account. A key issue is the definition of innovative theories, computational paradigms, linguistic mechanisms and implementation techniques for the design, realisation, deployment and management of global computational environments and their application.

A successful contribution to this research line is KLAIM, an experimental language with primitives for programming global computers that combines the process algebra approach with the coordination-oriented one. Its main features are process distribution and mobility, remote operations and asynchronous communication via multiple distributed tuple spaces. KLAIM has proved to be suitable for programming a wide range of distributed applications with agents and code mobility, and has been implemented on the top of a runtime system written in Java.

In this thesis, we first presents some foundational calculi for mobility based on KLAIM. Then, we concentrate on one of these calculi, namely μ KLAIM, that retains most of the peculiar features of KLAIM, while being much simpler. We present two approaches to study the behaviours of global computing systems expressed in μ KLAIM: (non-standard) *type systems* and *behavioural equivalences*. Type systems permit to control resource accesses, as well as data and process movements. Behavioural equivalences, on the other hand, permit to state and verify properties of distributed applications. Finally, we focus on the expressive power of the calculi by providing encodings of each calculus into a simpler one. The overall expressiveness is then assessed via a formal comparison with the asynchronous π -calculus.

Acknowledgements

In the years of my PhD, I had the luck of being supervised by *Rocco De Nicola* and *Rosario Pugliese*. Apart from their scientific support, that was really enormous, I could also benefit from their friendship, humanity and honesty. Their experience and their suggestions taught me the way theorems can be proved but, mainly, gave me the enthusiasm of pursuing research. I wish their teachings will always remain fixed in my mind. Rocco also allowed me to work in the group of Florence, while living in Rome: this was extremely important for me and for my family.

I am also incredibly indebted with *Anna Labella*, my MS supervisor. Her friendship and her support are at the basis of my PhD: she introduced me to Rocco and she always engaged me in collaborations at the Department of Informatics of the University of Rome “La Sapienza”. I also thank the deans of this department for the office given me in these years.

My research benefited from the collaborations with several, first-class authors. A part from Rocco and Rosario, I mention (in alphabetical order) *Michele Boreale*, *Chiara Braghin*, *Matthew Hennessy*, *Anna Labella*, *Vladimiro Sassone* and the *Concurrency and Mobility Group* in Florence. Most of the work carried on with them does not appear in this thesis; however, it strongly influenced all my research. In particular, Michele introduced me the field of semantics, while Matthew and Vladimiro very friendly hosted me in the Department of Informatics of the University of Sussex (Brighton, UK). Moreover, I also benefited from several discussions with a lot of people: Cedric Fournet (that kindly hosted me at Microsoft Research, Cambridge – UK), Jose Fiadeiro, Antonia Lopez and all the researchers involved in the projects we were part of.

I am very grateful to *Gérard Boudol* and *Michele Bugliesi* that referred this thesis. Their constructive attitude and the time they spent on these pages produced a lot of precious suggestions that improved the quality of my work. I also thank the members of my PhD jury for their efforts aimed at evaluating my work.

My research and my personal education required a lot of money, that were provided by several companies and institutions. First of all, I really thank the *Microsoft Research at Cambridge (UK)* that financed my PhD grant within the project *NAPI*. In these years, I had the opportunity to attend several conferences, schools and meetings. The money I spent were kindly provided by the EU and by the Italian MURST by means of several research projects: *Agile* (EU-FET-GC, Contract IST-2001-32747), *Mikado* (EU-FET-GC, Contract IST-2001-32222), *TOSCA* (MURST-Cofin) and *NAPOLI* (MURST-Cofin). The EU also paid my stay at Brighton in the summer 2003 with a *Marie Curie fellowship*.

Last but not least, I cannot describe the impact of my family in my work. My parents always supported my education with their love (and their money) and always encouraged me to take my own choices. My mother was a fundamental presence in our home and helped us even when her health suggested a rest. My children, Martina and Niccolò, filled my life of joy. Finally, my wife Monica is the essence of my happiness: she gave me the strength to react against the difficulties of everyday life, she had the patience of bearing me and she was always the first supporter of my works. This thesis is dedicated to all of them.

Contents

1	Introduction	9
1.1	Global Computers	9
1.2	Formal Methods for Global Computers	11
1.3	The Language KLAIM	13
1.4	Overview of the thesis	15
2	A Family of Process Languages	17
2.1	KLAIM	17
2.2	μ KLAIM: micro KLAIM	22
2.3	cKLAIM: core KLAIM	23
2.4	lcKLAIM: local core KLAIM	23
2.5	Related Work: Languages for GCs	24
3	Types to Control Process Activities	29
3.1	Overview: the Approach and Key Principles	30
3.2	A Basic Type System	32
3.2.1	Static Semantics	32
3.2.2	Dynamic Semantics	34
3.2.3	Type Soundness	37
3.3	Dynamic Management of Capabilities	41
3.3.1	Language Semantics	43
3.3.2	Type Soundness	47
3.3.3	Example: Subscribing On-line Publications	50
3.3.4	Variations on Capability Management	52
3.3.5	Discussion on Capability Management	63
3.4	Fine-grained Controls on Process Activities	65
3.4.1	Fine-grained Types	65
3.4.2	Static Semantics	68
3.4.3	Dynamic Semantics and Type Soundness	69
3.4.4	Example: A Bank Account Management System	73
3.5	Related Work	75

4	Types for Confining Data and Processes	79
4.1	Controlling Data Movement via Types	80
4.2	Static Inference and Checking	81
4.3	Typed Operational Semantics	85
4.4	Type Soundness	87
4.5	Example: Implementing a Multiuser System	91
4.6	Discussion and Related Work	96
5	Behavioural Theories	99
5.1	Touchstone Equivalences	100
5.2	Bisimulation Equivalence	104
5.2.1	Soundness w.r.t. Barbed Congruence	107
5.2.2	Completeness w.r.t. Barbed Congruence	113
5.3	Trace Equivalence	117
5.3.1	Soundness w.r.t. May Testing	120
5.3.2	Completeness w.r.t. May Testing	123
5.4	Verifying a Protocol for the Dining Philosophers	129
5.5	Equational Laws and the Impact of Richer Contexts	132
5.6	Related Work	134
6	Expressiveness of the Languages	137
6.1	Technical Preliminaries	138
6.2	KLAIM vs μ KLAIM	140
6.3	μ KLAIM vs cKLAIM	149
6.4	cKLAIM vs LcKLAIM	159
6.5	A Comparison with the π_a -calculus	163
6.5.1	Encoding the π_a -calculus in cKLAIM	164
6.5.2	Encoding LcKLAIM in the π_a -calculus	169
6.6	Concluding Assessment and Related Work	172
7	Conclusion and Future Work	177
A	Symbols and Notations	179

Chapter 1

Introduction

Technological advances of both computers and telecommunication networks, and development of more efficient communication protocols are leading to a ever-increasing integration of computing systems and to diffusion of the so called *global computer* (GC, for short). This is a massive networked and dynamically reconfigurable infrastructure interconnecting heterogeneous, typically autonomous and mobile components, that can operate on the basis of incomplete information.

A key research challenge is to devise theoretical models and calculi with a formal semantics for specifying, programming and reasoning about global computing systems. These calculi could provide a sound basis for constructing GCs which are “sound by construction” and which behave in a predictable and analysable manner. The crux is to identify what abstractions are more appropriate and to supply foundational and effective tools to support development and certification (stating and proving correctness) of global computing applications. This thesis aims at presenting a fully accounted work along this direction.

Before starting with the technical details, we shall carry on a more exhaustive discussion about the key features of GC and describe the state-of-the-art in this field. A summary of the thesis is given at the end of this chapter.

1.1 Global Computers

GC merge the features of traditional *distributed systems* together with the features of *open systems*. They borrow from distributed systems the intrinsic concurrency, the distribution of components, the absence of a global state and the asynchrony in changes of local states. They borrow from open systems the possibility for new entities (usually called *agents* or *processes*) of dynamically enter and exit the system, the heterogeneity and autonomy of components, the dynamic change of system configuration and the mobility of programs/data.

Applications for GC distinguish themselves from traditional distributed applications in terms of *scalability* (huge number of users and nodes), *connectivity* (both availability and bandwidth), *heterogeneity* (of operating systems and application

software), *autonomy* (of administration domains and mobile agents), and, mostly, in terms of the ability of dealing with *dynamic* and *unpredictable* changes of their network environment (e.g. availability of network connectivity, lack of resources, node failures, network reconfigurations and so on). GC are fostering a new style of distributed programming whose key principle is *network awareness*: applications now have information on network settings and can adapt to their variation. Indeed, applications often need to be aware of the administrative domains where they are currently located, and need to know how to cross administrative boundaries or how to access remote resources. Moreover, distant communication and use of remote resources is also affected, e.g., by the physical distance between locations and by congestion or (temporary/permanent/intermittent) failure of the underlying communication network [34].

The explicit introduction of locations in the picture imposes some sensible programming choices. Indeed, locations are abstractions for components (i.e., processes or resources) which share some interface with the environment. The kind of this interface may be, however, of various nature. For example, one can consider locations as *units of communication* (i.e. they provide address spaces for inter-processes exchanges), *units of migration* (i.e. co-located components move as a whole), *units of failure* (i.e. co-located components fail as a whole), *units of security* (i.e. co-located components share a security policy), and so on. Thus, every language for GCs must clarify the intended meaning assigned to locations; usually, the constructs of the language are then tailored accordingly.

The explicit knowledge of the distributed framework underlying a GC leads to another relevant consideration: programmers can refer localities when writing their programs. This fact requires the introduction in the language of constructs to deal with network awareness. Moreover, such constructs should be simple and powerful, while respecting the underlying meaning assigned to localities.

A suitable abstraction to design and program network-aware applications is *mobility*. Its usefulness emerges when developing applications that can implement both processes moving across the net to run over different hosts (*mobile computation*) and mobile devices with intermittent access to the network (*mobile computing*). In the literature, the term mobility is used to denote different mechanisms, ranging from simple ones (e.g., downloading of code [7]) to more sophisticated ones (supporting the migration of entire computations, e.g. [149, 2, 103]). Mobility has produced new interaction paradigms [73], like *Remote Evaluation*, *Code On-Demand* and *Mobile Agents*, that differ from traditional client-server patterns because they permit exchanging active units of behaviour and not just of raw data. These paradigms increase, e.g., network usage, fault-tolerance, service customisation, software maintenance and possibility of disconnected operations. They could be used, e.g., in distributed information retrieval, advanced telecommunication services (like video-on-demand), e-commerce and so on.

To support this programming style, new commercial/prototype programming languages with suitable constructs have been designed (e.g. Agent TCL [82], Facile [143], Java [7], Obliq [33], Pict [144, 127], TACOMA [97], Tele-

script/Odissey [148, 76]); this activity has involved several important ICT companies (e.g. Dec, General Magic, IBM, Microsoft, Sun) and academic research institutes. For a detailed analysis of several languages and calculi for mobility and for a taxonomical comparison between them, see [21].

1.2 Formal Methods for Global Computers

GC programming has prompted the study of the foundations of languages with advanced features, including mechanisms for process mobility, for coordinating and monitoring the use of resources, and for supporting the specification and the implementation of security policies. In particular, several specification and analysis techniques have been developed to build safe and trustworthy systems, to demonstrate their conformance to specifications, and to analyse their behaviour.

Foundational calculi have been used to supply formal foundations to the design of programming languages. A foundational calculus is both a kernel programming language and a computational model for describing and reasoning about the behaviour of programs. It supplies a formal basis to identify and generate new programming abstractions and analytical tools. A well-known example of foundational calculus for programming languages is the λ -calculus (and its variants) [8]. Several foundational languages, presented as process calculi or strongly based on them, have been developed that have improved the formal understanding of the complex mechanisms underlying network awareness and code mobility. We mention the Ambient calculus [41], the $D\pi$ -calculus [90], the DJoin calculus [71], the Seal calculus [45] and Nomadic Pict [140]. Their programming models encompass abstractions to represent the execution contexts where applications roam and run, and primitive mechanisms for inter-process communication and coordination.

The most useful theoretical mechanisms to express properties of concurrent systems are *type systems*, *behavioural equivalences* and (*modal*) *logics*. In this thesis we shall explore the first two approaches; a study on modal logics for KLAIM can be found in [106].

Type Systems. The idea of statically controlling the execution of a program via types dates back in time. The traditional property enforced by types, i.e. *type safety*, implies that every data will be used consistently with its declaration during the computation (e.g., an integer variable will always be assigned integer values). A similar approach has been incorporated in calculi for concurrent systems since their origins; we would like to mention, among the others, the seminal works in [112, 111, 152, 126, 136]. The basic typing principles presented in these works have been then enriched to enforce more sophisticated properties in [135, 101, 102, 99, 100, 92, 95, 94, 20]. Essentially, in all these work, types monitor the usage of communication media and, hence, well-typed programs are guaranteed to respect some expected property (e.g., absence of communication errors due to arity mismatching, liveness, linear usage of resources, or uniform receptiveness).

The type systems developed for GC languages (see [154, 128, 129, 88, 38, 105, 24, 25, 26, 43, 69, 59], among the others) evolve further to control *resource access* and *process mobility*. Types should mainly monitor the use of *communication media* and *migrations*. The main differences among the various type systems is the way typing information is exploited (only at compile time or, partially, also at run time) and the way it is stored in the system (centralised in some omniscient authority, split in several disjoint parts and assigned to different locations, partially split and partially shared between locations). In particular, we would like to remark some crucial general points.

- A statically type checked language for GC is definitely interesting from a theoretical point of view, but is quite unusable in practise. Indeed, the net is usually too large to allow a preliminary type checking of all its nodes or, even worse, not all nodes accept to be type checked (e.g. malicious nodes hosting viruses or misbehaving processes). Hence, a certain amount of run time overhead is necessary if we want to save the expressive power of the languages (indeed, one can easily imagine very strict syntactic rules that, even if protecting a net from misbehaviours, also reject legal nets). Nevertheless, global type checking can be used as a tool to ensure that partially well-typed systems work correctly.
- The presence of a unique typing context (an omniscient authority) allows for a greater number of static checks but it is quite unrealistic especially for GCs, where different administrators are responsible for the assignment of different policies. Thus, for the sake of realism, the typing information must be somehow split between the domains of the calculi (i.e. the nodes of the net). Again, maintaining some shared information simplifies and makes type checking more efficient, but it is not always a possible assumption.

Logics. An approach that is somehow related to types is the use of modal and temporal logics, i.e. standard first order logics equipped with reserved constructs to express *modalities*, i.e. properties of systems. The typical modalities used in concurrency (see, e.g., [86, 114]) express the ability of performing some kind of actions. This can be used, for example, for establishing deadlock freedom, liveness and correctness with respect to a given specification. As we already said, the same properties can be enforced with a type system. Indeed, like a type, a modal formula is satisfied by all the terms that enjoy the properties described by the formula. Differently from types, logics are ‘more abstract’ in that the formulae only focus on some crucial behaviour and ignores the remaining one, while types usually consider the overall term, not only some ‘sensible’ parts.

Logics take also advantage from the theory developed to automatically check the satisfaction of a formula, by exploiting tools for *model checking* [51, 141, 150].

In the setting of GC, some more advanced logics have been recently proposed in literature [40, 42, 32, 66] to establish such properties as resources allocation, access to resources, information disclosure and spatial allocation of processes/resources.

Behavioural Equivalences. Behavioural theories are a well-known and established tool to study system components in isolation, but compositionally [110, 65, 146, 147, 15, 115, 113, 96, 14]. Behavioural equivalences are usually exploited to abstract away a process from its syntactic structure and isolate the essence of its functionality. The theory can be used in several ways: to prove the soundness of a protocol implemented in the language, to prove some form of correspondence between a process written in a language and its encoding in another language, or to provide the theoretical foundation of an optimisation procedure (that, e.g., takes a process and produces a more efficient, but still functionally equivalent, one).

In order to be interesting, equivalences have to be *congruences*, i.e. they should be closed w.r.t. all the possible contexts of the language. A natural way to define congruences is via an explicit universal quantification over language contexts; however, this makes proving equivalences hard. Sometimes, this problem can be overcome by defining the operational semantics of the languages via a labelled transition system (LTS, for short), so that, when a system evolves, the action performed is made apparent. In this way, the interaction with an external context is recorded in the labels and the universal quantification can be dropped: equivalence proving is made tractable. Two well-known and studied (tractable) equivalences are *bisimulation* [123, 109] and *trace* [91, 65].

Developing equivalences for GCs is a non trivial task. Indeed, a GC usually requires higher-order and asynchronous communication paradigms that complicates the behavioural theory (see, e.g., [134, 137]). Bisimulation equivalences for some GC calculi have been recently appeared in literature [85, 107, 28, 44, 108]

1.3 The Language KLAIM

Several foundational languages, presented as process calculi or strongly based on them, have been developed that have improved the formal understanding of the complex mechanisms underlying GCs. In our view, a language for global computing should be equipped with primitives that support *network awareness* (i.e. locations can be explicitly referenced and operations can be remotely invoked), *disconnected operations* (i.e. code can be moved from one location to the other and remotely executed), *flexible communication mechanisms* (like distributed repositories [56, 49, 68] storing content addressable data), and *remote operations* (like asynchronous remote communications). Among the proposals appeared in literature in the last decade, we want to mention the Ambient calculus [41], $D\pi$ [85], DJoin [71] and Nomadic Pict [140]. They are languages equipped with primitives to represent at various abstraction levels the execution contexts of the net where applications roam and run, they provide mechanisms for coordinating and monitoring the use of resources, and they support the specification and the implementation of security policies. However, if one contrasts them with the above list of distinguishing features of languages for GCs, one realizes that all of them fall short for at least one of the targets.

KLAIM (*Kernel Language for Agents Interaction and Mobility*, [57]) is an experimental language with programming constructs for GCs that combines the process algebraic paradigm with the coordination-oriented one and that satisfies all the requirements we mentioned in the previous paragraph. It rests on an extension of the basic Linda coordination model [74] with multiple distributed tuple spaces. A *tuple space* is a multiset of *tuples* that are sequences of information items. Tuples are anonymous and associatively selected from tuple spaces by means of a *pattern-matching* mechanism. The Linda model was originally proposed for parallel programming on isolated machines. Multiple, possibly distributed, tuple spaces have been advocated later [75] to improve modularity, scalability and performance. The obtained communication model has a number of properties that make it appealing for GCs (see, e.g., [56, 49, 68]). The model permits *time uncoupling* (data life time is independent of the producer process life time), *destination uncoupling* (the producer of a datum does not need to know the future use or the destination of that datum) and *space uncoupling* (communicating processes need to know a single interface, i.e. the operations over the tuple space). As shown in [68], where several messaging models for mobile processes are examined, the *blackboard* approach, of which tuple spaces are variants, is one of the most appreciated, also because of its flexibility. Evidence of the success gained by the tuple space paradigm is given by the many tuple space based run-time systems, both from industries (JavaSpaces [6] and IBM TSpaces [151]) and from universities (PageSpace [50], WCL [131], Lime [125] and TuCSon [121]).

KLAIM handles multiple distributed tuple spaces that are placed on nodes of a net. The nodes of a net can be thought of as physically distributed machines, or as logical partitions of the same machine, or, broadly speaking, as shared resources. Each node can be accessed through its locality and contains a single tuple space and processes in execution. Localities can be dynamically created and are handled via sophisticated scoping rules *à la* π -calculus. Processes can be executed concurrently both at the same node or at different nodes and can perform a few basic operations over tuple spaces and nodes: retrieve/place tuples from/into a tuple space, send processes for execution on (possibly remote) nodes, and create new nodes. Interprocess communication is *asynchronous*: the producer (i.e. sender) and the consumer (i.e. receiver) of a tuple do not need to synchronise.

KLAIM has a rich set of constructs that ease the task of programming and are at the basis of the programming language X-KLAIM [11, 9], whose run-time system [12, 52] is written in Java. The features it offers have proved to be suitable for programming a wide range of distributed applications with agents and code mobility [57, 58] that, once compiled in Java, can be run over different platforms.

One significant design choice underlying KLAIM is the possibility of abstracting from the exact physical allocation of some resources in a net. Indeed, differently from most process languages, KLAIM also considers for execution open processes, i.e. processes containing unbound *variables* (that are traditionally considered as programming errors). Unbound variables can be thought of as the symbolic names for physical addresses (i.e., *localities*). The bindings between variables and locali-

ties are stored in the *allocation environment* of each node of the net. Thus, when a (open) process uses an unbound variable, the variable is translated to a locality via the allocation environment of the node where the action is executed. In this way, programmers are not required to know the precise structure of the whole net; they can structure programs over distributed environments while ignoring the precise allocation of some resources.

KLAIM-derived Calculi. At least, two possible critiques can be moved to KLAIM, that somehow contrast each other.

- The design choices described above make the language quite heavy both in the static and in the dynamic semantics (see the type systems and the operational semantics of [57, 58, 59]). Thus, no behavioural theory has ever been developed and, more generally, *it can be hardly considered as a process calculus*.
- On the other hand, *it is also quite far from a real programming language* in that it lacks standard constructs like ‘if-then-else’, ‘while-do’, ‘for’ and so on. Moreover, no default data type is provided (except from records, i.e. tuples); thus, every kind of data structure and all the operations they usually provide must be explicitly implemented.

In [9], the second problem has been addressed and a fully-fledged programming language, X-KLAIM, has been introduced. Its run-time system is written in Java and, thus, can run over several kinds of platforms. This thesis aims to remedy to the first critique. We distill from the KLAIM paradigm some basic calculi; then, we develop on them simple but meaningful type systems and semantic theories.

More precisely, in the next Chapter we present three calculi derived from KLAIM. The first one is μ KLAIM (*micro-KLAIM*) [80]. Mainly, it is obtained by removing the allocation environments from KLAIM’s syntax. Thus, by following the π -calculus, we can further simplify the language and assume just one syntactic category of *names* (instead of distinguishing between localities and variables). A second simplification step yields cKLAIM (*core KLAIM*) [10]. It is obtained from μ KLAIM by removing the primitive **read** and by considering only monadic data (i.e. tuples with only one field). Finally, by also excluding the possibility of remote communications, we obtain lcKLAIM (*local-cKLAIM*) [62]. In the latter calculus, the only remote operation is code migration; thus, it is very similar to $D\pi$, except for the fact that the communication is asynchronous and based on a shared memory paradigm.

1.4 Overview of the thesis

“Thesis of this thesis” The work we shall present here lays the semantic foundations of the KLAIM language. To this aim, we isolate a kernel calculus for KLAIM that retains most of the expressive power of the original language, and use it to formally study the semantic and type theoretic basis of the considered model.

Structure of the thesis In Chapter 2 we formally present KLAIM , i.e. its syntax and operational semantics (via a structural equivalence and a reduction relation). A simple programming example is then given to illustrate the usability of the language features. Then, we formally present the three calculi μKLAIM , cKLAIM and lcKLAIM ; in particular, μKLAIM will be the elected reference calculus.

In Chapter 3 we present type systems to implement resource access and mobility control. We start with a very basic type system that controls the (local and remote) operations a process wants to perform when running in a given node. Then, we tune the basic setting to encompass more involved features, like dynamic modification of policies and a fine-grained control over the legal operations. This Chapter is based on [80, 79]. In Chapter 4 we present another typing approach to control the movement of data and processes, as presented in [81, 61]. Data are tagged with a set of localities and they can cross only nodes whose addresses are in their tag. The execution of processes is then constrained accordingly.

In all the typing theories we shall present in this work, we follow an approach that mixes static and dynamic checks. This choice will be motivated in Chapter 3. We want to anticipate that completely static disciplines can be developed: e.g., in [61] we adapt the confining types presented in Chapter 4 to $D\pi$ and to (a variant of) the Ambient calculus, where static typing is only used. However, static typings conflict with the principles underlying GC and tuple-spaces; thus, we are forced to use more dynamic techniques.

In Chapter 5 we turn our attention to the behavioural semantics of our languages. To this aim, we first define a *reduction barbed congruence* and a *may-testing* equivalence. As usual, such congruences rely on an universal quantification over language contexts and, thus, are difficult to handle. We define a labelled transition system as an alternative (but equivalent) operational semantics and build up over it non standard (tractable) bisimulations and trace equivalences. The theory is then exploited to prove properties of a well-known distributed protocol, namely the “Dining philosophers”. The work of this Chapter is an adaption of [60].

In Chapter 6 we will see that μKLAIM is a very good candidate to be *the* kernel calculus underlying KLAIM . Indeed, by means of a few encodings, we shall show that μKLAIM is a good compromise between the expressive power of KLAIM and that of a more basic process calculus, like the asynchronous π -calculus. By examining the properties enjoyed by the encodings, we shall also evaluate the impact of some design issues underlying our calculi. These results are based on [62].

In Chapter 7 we conclude the thesis and show possible directions for future work.

Chapter 2

A Family of Process Languages

We now formally present the languages we shall work with, namely KLAIM [57] and three calculi derived from it. The first one is μKLAIM , where, essentially, the distinction between logical and physical names has been removed from KLAIM . From μKLAIM , by only considering monadic communications and by removing the action **read**, we obtain cKLAIM . Finally, by also removing the possibility of performing remote inputs/outputs (thus, by only relying on migration for using remote resources) we obtain lcKLAIM .

2.1 KLAIM

The syntax of KLAIM is given in Table 2.1. We assume two disjoint countable sets: \mathcal{L} of *locality names* l, l', \dots and \mathcal{V} of *variables* $x, y, \dots, X, Y, \dots, \text{self}$, where **self** is a reserved variable (see below). Notationally, we prefer letters X, Y, \dots when we want to stress the use of a variable as a process variable and x, y, \dots otherwise. We will use u for basic variables and localities.

Processes, ranged over by P, Q, R, \dots , are the KLAIM active computational units and may be executed concurrently either at the same locality or at different localities. Processes are built from the terminated process **nil** and from basic actions by using action prefixing, parallel composition and recursion. *Basic Actions*, ranged over by a , permit removing/accessing/adding data from/to node repositories, activating new threads of execution and creating new nodes. Action **new** is not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take effect. *Tuples*, t , are the communicable objects: they are sequences of names and processes. *Templates*, T , are patterns used to retrieve tuples and the pattern matching underlying the communication mechanism is the one used for LINDA [74].

Nets, ranged over by N, M, H, K, \dots , are finite collections of nodes. A *node* is a triple $l ::_{\rho} C$, where locality l is the address of the node, ρ is the *allocation environment* (a finite partial mapping from variables to names, used to implement dynamic binding of names) and C is the component located at l . *Components*,

$N ::= \mathbf{0}$	$ $	$l ::=_{\rho} C$	$ $	$N_1 \parallel N_2$	$ $	$(\nu l)N$		
$C ::= \langle t \rangle$	$ $	P	$ $	$C_1 C_2$				
$P ::= \mathbf{nil}$	$ $	$a.P$	$ $	$P_1 P_2$	$ $	X	$ $	$\mathbf{rec} X.P$
$a ::= \mathbf{in}(T)@u$	$ $	$\mathbf{read}(T)@u$	$ $	$\mathbf{out}(t)@u$	$ $	$\mathbf{eval}(P)@u$	$ $	$\mathbf{new}(l)$
$t ::= u$	$ $	P	$ $	t_1, t_2				
$T ::= u$	$ $	$!x$	$ $	$!X$	$ $	T_1, T_2		

Table 2.1: KLAIM syntax

ranged over by C, D, \dots , can be either processes or data, denoted by $\langle t \rangle$. In the net $(\nu l)N$, the scope of the name l is restricted to N ; the intended effect is that if one considers the net $N_1 \parallel (\nu l)N_2$ then locality l of N_2 cannot be immediately referred to from within N_1 . We say that a net is *well-formed* if for each node $l ::=_{\rho} C$ we have that $\rho(\mathbf{self}) = l$, and, for any pair of nodes $l ::=_{\rho} C$ and $l' ::=_{\rho'} C'$, we have that $l = l'$ implies $\rho = \rho'$. Hereafter, we will only consider well-formed nets.

Names and variables occurring in KLAIM processes and nets can be *bound*. More precisely, prefix $\mathbf{new}(l).P$ binds l in P , and, similarly, net restriction $(\nu l)N$ binds l in N . Prefix $\mathbf{in}(\dots, !_-, \dots)@u.P$ binds variable $_$ in P ; this prefix is similar to the λ -abstraction of the λ -calculus. Finally, $\mathbf{rec} X.P$ binds variable X in P . A name/variable that is not bound is called *free*. The sets $fn(\cdot)$ and $bn(\cdot)$ (respectively, of free and bound names of a term) and $fv(\cdot)$ and $bv(\cdot)$ (of free/bound variables) are defined accordingly. The set $n(\cdot)$ is the union of the free and bound names and variables occurring in \cdot . As usual, we say that two terms are *alpha-equivalent*, written $=_{\alpha}$, if one can be obtained from the other by renaming bound names/variables. We shall say that u is *fresh* for $_$ if $u \notin n(\cdot)$. In the sequel, we shall work with terms whose bound variables are all distinct and whose bound names are all distinct and different from the free ones.

Remark 2.1.1 The language presented so far slightly differs from [57]: the two differences are the absence of values and expressions, and the use of recursion instead of process definitions. Values and expressions (e.g., integers, strings, ...) are not included only to simplify reasoning, while recursion is easier to deal with in a theoretical framework (the syntax of a recursive term already contains all the code needed to properly run the term itself).

Notations and Conventions. We write $A \triangleq W$ to mean that A is of the form W ; this notation is used to assign a symbolic name A to the term W . We shall use notation \tilde{u} to denote sequences of names or variables; this will be sometimes written as $\tilde{u}_{i \in I}$, for an appropriate index-set I . Moreover, if $\tilde{u} = (u_1, \dots, u_n)$, we shall assume that $u_i \neq u_j$ for $i \neq j$. If $\tilde{u}_1 = (u_1^1, \dots, u_n^1)$ and $\tilde{u}_2 = (u_1^2, \dots, u_m^2)$ then \tilde{u}_1, \tilde{u}_2 will denote the sequence of pairwise distinct elements $(u_1^1, \dots, u_n^1, u_1^2, \dots, u_m^2)$. When convenient, we shall regard a sequence simply as a set.

We shall sometimes write $\mathbf{in}()@l$, $\mathbf{out}()@l$ and $\langle \rangle$ to mean that the argument of

(STR-PZERO)	$N \parallel \mathbf{0} \equiv N$	
(STR-PCOM)	$N_1 \parallel N_2 \equiv N_2 \parallel N_1$,	
(STR-PASS)	$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$	
(STR-ALPHA)	$N \equiv N'$	if $N =_\alpha N'$
(STR-RCOM)	$(\nu l_1)(\nu l_2)N \equiv (\nu l_2)(\nu l_1)N$	
(STR-EXT)	$N_1 \parallel (\nu l)N_2 \equiv (\nu l)(N_1 \parallel N_2)$	if $l \notin \text{fn}(N_1)$
(STR-ABS)	$l ::_\rho C \equiv l ::_\rho (C \mid \mathbf{nil})$	
(STR-CLONE)	$l ::_\rho C_1 \mid C_2 \equiv l ::_\rho C_1 \parallel l ::_\rho C_2$	
(STR-REC)	$l ::_\rho \mathbf{rec} X.P \equiv l ::_\rho P[\mathbf{rec} X.P/X]$	

Table 2.2: KLAIM Structural Equivalence

the actions or the datum are an empty sequence of items. We usually omit trailing occurrences of process **nil** and write $\Pi_{j \in J} W_j$ for the parallel composition (both ‘|’ and ‘||’) of terms (components or nets, resp.) W_j . Similarly, we write $\{\dots\}_{j \in J}$ to mean $\bigcup_{j \in J} \{\dots\}$.

We also assume that allocation environments act as the identity on locality names. This assumption simplifies the operational semantics.

Finally, for the sake of readability, in the examples we will omit trailing occurrences of process **nil**, and use parameterised process definitions (that can be easily implemented in our setting using **out/in** operations to pass/recover the parameters). Also, some kind of basic values, like integers and strings, will be silently assumed. They can be implemented by following [112].

The operational semantics relies on a *structural congruence* relation, \equiv , bringing the participants of a potential interaction to contiguous positions, and a *reduction relation*, \mapsto , expressing the evolution of a net. The structural congruence is the least congruence closed under the axioms given in Table 2.2. Most of the rules are standard [112], while laws (STR-ABS) and (STR-CLONE) are peculiar to our setting. The first one states that **nil** is the identity for ‘|’; the second one turns a parallel between co-located components into a parallel between nodes (thus, it is also used to achieve commutativity and associativity of ‘|’). The reduction relation is given in Table 2.3, where we use two auxiliary functions:

1. a *tuple/template evaluation* function, $\mathcal{E}[_]_\rho$, to transform variables according to the allocation environment of the node performing the action whose argument is $_$. The main clauses of its definition are given below:

$$\mathcal{E}[_]_\rho = \begin{cases} u & \text{if } u \in \mathcal{L} \\ \rho(u) & \text{if } u \in \text{dom}(\rho) \\ \text{UNDEF} & \text{otherwise} \end{cases} \quad \mathcal{E}[P]_\rho = P\{\rho\}$$

where $P\{\rho\}$ denotes the process obtained from P by replacing any free occurrence of a variable x that is not within the argument of an **eval** with $\rho(x)$.

(RED-OUT)	$\frac{\rho(u) = l' \quad \mathcal{E}[\![t]\!]_{\rho} = t'}{l ::_{\rho} \mathbf{out}(t)@u.P \parallel l' ::_{\rho'} \mathbf{nil} \mapsto l ::_{\rho} P \parallel l' ::_{\rho'} \langle t' \rangle}$
(RED-EVAL)	$\frac{\rho(u) = l'}{l ::_{\rho} \mathbf{eval}(P_2)@u.P_1 \parallel l' ::_{\rho'} \mathbf{nil} \mapsto l ::_{\rho} P_1 \parallel l' ::_{\rho'} P_2}$
(RED-IN)	$\frac{\rho(u) = l' \quad \mathit{match}(\mathcal{E}[\![T]\!]_{\rho}, t) = \sigma}{l ::_{\rho} \mathbf{in}(T)@u.P \parallel l' ::_{\rho'} \langle t \rangle \mapsto l ::_{\rho} P\sigma \parallel l' ::_{\rho'} \mathbf{nil}}$
(RED-READ)	$\frac{\rho(u) = l' \quad \mathit{match}(\mathcal{E}[\![T]\!]_{\rho}, t) = \sigma}{l ::_{\rho} \mathbf{read}(T)@u.P \parallel l' ::_{\rho'} \langle t \rangle \mapsto l ::_{\rho} P\sigma \parallel l' ::_{\rho'} \langle t \rangle}$
(RED-NEW)	$l ::_{\rho} \mathbf{new}(l').P \mapsto (v l')(l ::_{\rho} P \parallel l' ::_{\rho[l'/\text{se1f}]} \mathbf{nil})$
(RED-PAR)	$\frac{N_1 \mapsto N'_1}{N_1 \parallel N_2 \mapsto N'_1 \parallel N_2}$
(RED-RES)	$\frac{N \mapsto N'}{(v l)N \mapsto (v l)N'}$
(RED-STRUCT)	$\frac{N \equiv M \mapsto M' \equiv N'}{N \mapsto N'}$

Table 2.3: KLAIM Reduction Relation

Clearly, $\mathcal{E}[\![P]\!]_{\rho}$ is UNDEF if $\rho(x)$ is undefined for some of these x .¹ We shall write $\mathcal{E}[\![t]\!]_{\rho} = t'$ to denote that the evaluation of t using ρ succeeds and returns t' .

2. a *pattern matching* function, $\mathit{match}(\cdot, \cdot)$, to verify the compliance of a tuple w.r.t. a template and to associate values (i.e. names and processes) to variables bound in templates. Intuitively, a tuple matches a template if they have the same number of fields, and corresponding fields match (where a bound name matches any value, while two names match only if they are identical). Formally, function match is defined by the following rules:

$$\begin{aligned} \mathit{match}(l, l) &= \epsilon & \mathit{match}(!x, l) &= [!x] \\ \mathit{match}(!X, P) &= [P/X] & \frac{\mathit{match}(T_1, t_1) = \sigma_1 \quad \mathit{match}(T_2, t_2) = \sigma_2}{\mathit{match}(T_1, T_2, t_1, t_2) = \sigma_1 \circ \sigma_2} \end{aligned}$$

¹The definition of $\mathcal{E}[\![P]\!]_{\rho}$ given here slightly differs from the definition in [57]. There, $\mathcal{E}[\![P]\!]_{\rho}$ always succeeds since it leaves unresolved the x such that $\rho(x) = \text{UNDEF}$. In [57], processes with unresolved variables can occur as tuple fields (their free variables can be resolved successively) but their execution gets stuck when trying to perform an action involving unresolved variables. In the definition given here, unresolved processes cannot occur within evaluated tuples; this simplifies the work presented here without radically affecting the principles underlying KLAIM.

where we let ‘ ϵ ’ to be the empty substitution and ‘ \circ ’ to denote substitutions composition. Here, a substitution σ is a mapping of names and processes for variables; $P\sigma$ denotes the (capture avoiding) application of σ to P . Moreover, we assume that $P\sigma$ yields a process written according to the syntax of Table 2.1.

The intuition behind the operational rules of KLAIM is the following. In rule (RED-OUT), the local allocation environment is used both to determine the name of the node where the tuple must be placed and to evaluate the argument tuple. This implies that if the argument tuple contains a field with a process, the corresponding field of the evaluated tuple contains the process resulting from the evaluation of its free variables. Hence, processes in a tuple are transmitted after the interpretation of their free variables through the local allocation environment. This corresponds to a *static scoping* discipline for the (possibly remote) generation of tuples. A *dynamic linking* strategy is adopted for the **eval** operation, rule (RED-EVAL). In this case the free variables of the spawned process are not interpreted using the local allocation environment: the linking of variables is done at the remote node. Rules (RED-IN) and (RED-READ) require existence of a matching datum in the target node. The tuple is then used to replace the free occurrences of the variables bound by the template in the continuation of the process performing the actions. With action **in**, the matched datum is consumed while with action **read** it is not. Finally, in rule (RED-NEW), the environment of a new node is derived from that of the creating one with the obvious update for the `self` variable. Therefore, the new node inherits all the bindings of the creating node.

Notice that, even if there exist prefixes for placing data to nodes, no synchronization takes place between (sending and receiving) processes: hence, the communication paradigm is really asynchronous. On the contrary, a sort of synchronization takes place between a sending process and its target node (see rules (RED-OUT) and (RED-EVAL)). A similar synchronization takes place between the node hosting a datum and the process looking for it (see rules (RED-IN) and (R-MATCH)).

Remark 2.1.2 The main characteristic of KLAIM’s communication mechanisms is the possibility of retrieving data while partially analysing them (by exploiting the pattern matching function). As we shall see in Chapter 6, the pattern matching is very powerful and expressive. Nevertheless, its distributed implementation is quite lightweight: in X-KLAIM [9], the process performing an **in/read** first retrieves the tuple; it then performs the pattern matching locally and, in case of failure, restores the tuple at its original place.

Programming in KLAIM: A printing service. To illustrate KLAIM’s paradigm and its usefulness to implement distributed applications with mobile code, we now give a simple example. We suppose to have two departments modelled as network nodes with addresses `dep1` and `dep2`, respectively. We also have one printer associated to each of them; this is again modelled as two distinct network nodes with address

prnt_1 and prnt_2 . The logical name $print$ is resolved both by ρ_1 and by ρ_2 , the allocation environments of the departments; as expected, we let $\rho_1(print) = \text{prnt}_1$ and $\rho_2(print) = \text{prnt}_2$. Moreover, we let $\rho_1(\text{self}) = \text{dep}_1$ and $\rho_2(\text{self}) = \text{dep}_2$.

The printers are installed by running the following code:

$$\text{Inst} \triangleq \langle \text{start-up code} \rangle .\text{out}(\text{"printer address :"}, \text{print})@\text{self}$$

This process first executes some start-up activity and then, once the printer has been successfully installed, it publishes its address. If executed at dep_i (for $i = 1, 2$), the resulting net will be

$$\begin{aligned} & \text{dep}_1 ::_{\rho_1} \dots | \langle \text{"printer address :"}, \text{prnt}_1 \rangle \parallel \text{prnt}_1 :: \dots \\ & \parallel \text{dep}_2 ::_{\rho_2} \dots | \langle \text{"printer address :"}, \text{prnt}_2 \rangle \parallel \text{prnt}_2 :: \dots \end{aligned}$$

This reductions emphasise the use of the *local* allocation environment when performing actions **out**. Now, a client can send papers for printing in two ways:

1. he can retrieve (from a possibly remote node) the address of the printer of the first department and directly send his paper to the printer. This is implemented as

$$\text{RemPrint} \triangleq \text{read}(\text{"printer address :"}, !y)@\text{dep}_1 . \text{out}(\text{"print"}, \text{paper})@y$$

2. he can spawn a process that locally requires the print. This can be implemented as

$$\text{LocPrint} \triangleq \text{eval}(\text{out}(\text{"print"}, \text{paper})@\text{print})@\text{dep}_1$$

This solution relies on the *dynamic* handling of logical names; indeed, the name $print$ is resolved only after migration by exploiting the association $[print \mapsto \text{prnt}_1]$ in ρ_1 .

2.2 μKLAIM : micro KLAIM

The calculus μKLAIM has been derived in [80] from KLAIM by removing allocation environments and the possibility of having pieces of code as tuple fields.² Its syntax is given in Table 2.4. The removal of allocation environments makes it possible to merge together names and variables. Thus, we only assume a countable set \mathcal{N} of *names* $l, l', \dots, u, \dots, x, y, \dots, X, Y, \dots$. Names provide the abstract counterpart of the set of *communicable* objects and can be used as localities, basic variables or process variables: we do not need to distinguish between these three kinds of objects anymore. Like before, we prefer letters l, l', \dots when we want to stress the use of a name as a locality, x, y, \dots when we want to stress the use of a name as a

$N ::= \mathbf{0}$		$l :: C$		$N_1 \parallel N_2$		$(\nu l)N$
$C ::=$	like in Table 2.1					
$P ::=$	like in Table 2.1					
$a ::=$	like in Table 2.1					
$t ::= u$		t_1, t_2				
$T ::= u$		$!x$		T_1, T_2		

Table 2.4: μ KLAIM Syntax

basic variable, and X, Y, \dots when we want to stress the use of a name as a process variable. We will use u for basic variables and localities.

Notice that μ KLAIM can be considered as the largest sub-calculus of KLAIM where tuples do not contain any process, allocation environments are empty and all processes are closed. These modifications sensibly simplifies the operational semantics of the language. The structural congruence is readily adapted from Table 2.2; the key laws to define the reduction relation are given in Table 2.5. Notice that now tuples/templates evaluation function is useless and substitutions are (standard) mappings of names for names. Hence, the definition of function *match* is given by the following laws:

$$\begin{array}{l}
 \text{match}(l, l) = \epsilon \\
 \text{match}(!x, l) = [l/x] \\
 \text{match}(T_1, t_1) = \sigma_1 \quad \text{match}(T_2, t_2) = \sigma_2 \\
 \hline
 \text{match}(T_1, T_2, t_1, t_2) = \sigma_1 \circ \sigma_2
 \end{array}$$

2.3 cKLAIM: core KLAIM

The calculus cKLAIM has been introduced in [10] by eliminating from μ KLAIM action **read** and by only considering monadic communications (i.e. tuples and templates containing only one field). The formal syntax of cKLAIM is given in Table 2.6. Notice that cKLAIM is a sub-calculus of μ KLAIM and thus it inherits from μ KLAIM the operational semantics.

2.4 lCKLAIM: local core KLAIM

lCKLAIM is the version of cKLAIM where actions **out** and **in** can be only performed locally, i.e. the only remote primitive is action **eval** (this is the principle underlying the language $D\pi$, [90]). The syntax of the new calculus can be derived from the syntax of cKLAIM given in Table 2.6 by using the following production for process actions:

$$a ::= \mathbf{in}(T) \quad | \quad \mathbf{out}(t) \quad | \quad \mathbf{eval}(P)@u \quad | \quad \mathbf{new}(l)$$

²The calculus used here slightly differs from the calculus given in [80]: the differences are the absence of values and expressions (to simplify reasoning) and the use of recursion. These simplifications have been motivated in Remark 2.1.1.

(RED-OUT)	$l :: \mathbf{out}(t)@l'.P \parallel l' :: \mathbf{nil} \mapsto l :: P \parallel l' :: \langle t \rangle$
(RED-EVAL)	$l :: \mathbf{eval}(P_2)@l'.P_1 \parallel l' :: \mathbf{nil} \mapsto l :: P_1 \parallel l' :: P_2$
(RED-IN)	$\frac{\text{match}(T, t) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle \mapsto l :: P\sigma \parallel l' :: \mathbf{nil}}$
(RED-READ)	$\frac{\text{match}(T, t) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle t \rangle \mapsto l :: P\sigma \parallel l' :: \langle t \rangle}$
(RED-NEW)	$l :: \mathbf{new}(l').P \mapsto (\nu l')(l :: P \parallel l' :: \mathbf{nil})$
and rules (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 2.3	

Table 2.5: μ KLAIM Reduction Rules

We want to remark³ that lcKLAIM is a sub-calculus of cKLAIM : indeed, it is the largest sub-calculus of cKLAIM closed under the predicate \rightsquigarrow , defined as

$$\begin{aligned}
N \rightsquigarrow &\triangleq N = \mathbf{0} \quad \vee \quad (N = (\nu l)N' \wedge N' \rightsquigarrow) \quad \vee \\
&\quad (N = N_1 \parallel N_2 \wedge N_1 \rightsquigarrow \wedge N_2 \rightsquigarrow) \quad \vee \quad (N = l :: C \wedge C \rightsquigarrow_l) \\
C \rightsquigarrow_l &\triangleq C = \langle l' \rangle \quad \vee \quad (C = P \wedge P \rightsquigarrow_l) \quad \vee \\
&\quad (C = C_1 | C_2 \wedge C_1 \rightsquigarrow_l \wedge C_2 \rightsquigarrow_l) \\
P \rightsquigarrow_u &\triangleq (P = \mathbf{nil}, X) \quad \vee \quad (P = \mathbf{eval}(Q)@v.R \wedge Q \rightsquigarrow_v \wedge R \rightsquigarrow_u) \quad \vee \\
&\quad (P = P_1 | P_2 \wedge P_1 \rightsquigarrow_u \wedge P_2 \rightsquigarrow_u) \quad \vee \\
&\quad (P = \mathbf{in}(T)@u.Q, \mathbf{out}(t)@u.Q, \mathbf{new}(l).Q, \mathbf{rec} X.Q \wedge Q \rightsquigarrow_u)
\end{aligned}$$

The only relevant cases are those for prefixes **in/out/eval**: they ensure that actions **in** and **out** only specify as target node the node where the action is executed (i.e. the u decorating \rightsquigarrow_u).

The operational semantics of lcKLAIM is obtained by replacing rules (RED-OUT) and (RED-IN) of Table 2.5 with the following ones:

$$\begin{aligned}
(\text{RED-OUT}) \quad & l :: \mathbf{out}(l').P \mapsto l :: P | \langle l' \rangle \\
(\text{RED-IN}) \quad & l :: \mathbf{in}(T).P | \langle l' \rangle \mapsto l :: P\sigma \quad \text{if } \text{match}(T, l') = \sigma
\end{aligned}$$

2.5 Related Work: Languages for GCs

To conclude this chapter, we shall describe some of the most successful calculi including features for code distribution and mobility. We present the languages by grouping them according to the underlying design principles. Thus, we start with some distributed versions of the CCS [110] and of the π -calculus [113]; then, we

³This allows us to put an arrow ' \hookrightarrow ' between lcKLAIM and cKLAIM in Table 6.8 (see Chapter 6).

N	::=	like in Table 2.4
C	::=	like in Table 2.4
P	::=	like in Table 2.4
a	::=	$\mathbf{in}(T)@u$ $\mathbf{out}(t)@u$ $\mathbf{eval}(P)@u$ $\mathbf{new}(l)$
t	::=	u
T	::=	u $!x$

Table 2.6: cKLAIM Syntax

present the Ambient calculus [41] and other calculi derived from it. Here we only comment on the design choices underlying the various languages. More technical comments on the typing theories and behavioural semantics for (some of) these calculi will be given at the end of Chapters 3, 4 and 5.

Distributed versions of CCS

In the eighties and nineties, many CCS-like process calculi have been enriched with localities to explicitly describe the distribution of processes [47, 54, 22, 46]. The aim was mainly to provide these calculi with non interleaving semantics or, at least, to differentiate processes' parallel components (thus obtaining more in-spective semantics than the interleaving one). This line of research is far from the principles of GC, where localities are used as a mean to make processes network aware, thus enabling them to refer the network locations as target of remote communications or as destination of migrations. As we already said, localities in GCS are not only considered as units of distribution but, according to the case, as units of mobility, of communication, of failure or of security.

A more recent CCS-based calculus is [130]. There, processes run over the nodes of an explicit, flat and dynamically evolving net. Nodes can fail thus causing loss of all hosted processes. There are explicit operations to kill nodes and to query the status of a node; thus, failures can be detected. The operational semantics uses information on the state of nodes (either failed or alive), but it is otherwise very close to that of CCS. The idea is that distribution is transparent in absence of failures.

Distributed versions of the π -calculus

$D\pi$ [90]. $D\pi$ was firstly introduced in [128]. The language was equipped with complex features like dynamically evolving, hierarchically structured nets and primitives for moving part of the hierarchy, for moving code and for killing alive locations. The language was also equipped with a primitive to test the state of locations, thus enabling failure detection. The original language has been simplified in [90]; the calculus contains primitives for code movement and creation of new localities/channels in a net with a flat architecture. Communication occurs only between co-located processes; thus, processes must move to communicate.

D $\pi\lambda$ [153, 154]. *D $\pi\lambda$* enhances *D π* by integrating the call-by-value λ -calculus and the π -calculus, together with primitives for process distribution and remote process creation. The communication is higher-order, in that process code can be transmitted and retrieved over channels. Localities are anonymous (i.e. not explicitly referrable by processes) and simply used to express process distribution. Their function is to allow the development of fine-grained typing theories (this aspect will be illustrated and discussed in Chapter 3). This design choice, together with the absence of a migration primitive, makes *D $\pi\lambda$* unsuited for programming GCS. Indeed, it better models traditional distributed and multi-threaded systems, where distribution is transparent.

$\pi_{1\ell}$ -calculus [4]. The *$\pi_{1\ell}$ -calculus* extends the π -calculus to encompass distribution and mobility. Locations can host processes, can asynchronously fail and can be killed by other processes. The language enables creation of new locations and channels, permits testing for liveness of locations and supports code movement. A channel c allocated at ℓ is accessed simply by naming c , provided that name c is known and ℓ is alive.

DJoin [71]. In the Distributed Join calculus, located mobile processes are hierarchically structured and form a tree-like structure evolving during the computation. Entire subtrees, and not only single processes, can move. Technically, nets are flat collections of named nodes, where the name of a node indicates the nesting path; e.g., a node whose name is $l_1 \cdots l_k.l$ represents a node referrable to via the unique name l and that is nested in l_k , that is a node contained in l_{k-1} and so on. Communication in *DJoin* takes place in two steps: firstly, the sending process sends a message on a channel; then, the ether (i.e. the environment containing all the nodes) delivers the message to the (unique) process that can receive on that channel. Failures are modelled by tagging locality names: e.g. the (compound) name $\cdots l_i^\Omega \cdots l$ states that l is a node contained in a failed node l_i and, thus, l itself is failed. The Ω at l_i has been caused by execution of the primitive *halt* by a process running at l_i . Failures can be detected by using the primitive *fail*. Failed nodes cannot host running computations but can receive data/code/sublocations that, however, once arrived in the failed node, become definitely stuck.

dpi [139]. It is a distributed process calculus similar to the *DJoin* in that it combines the channel-based communication mechanism of the π -calculus with the hierarchical organisation and mobility of localities. However, differently from the *DJoin*, channels are not explicitly allocated by the syntax and no notion of failure is present. Channels can be used remotely or locally; a channel is local if it is accessed only by co-located processes.

Nomadic Pict [140, 145]. It is a distributed and agent-based version of *Pict* [127], a concurrent language based on the asynchronous π -calculus [93, 19]. The language relies on a net (a collection of named sites) where *named agents* can roam.

Both agents and sites are uniquely named. Channels are not located, but communication between two agents can take place only if they are located at the same node (thus no low-level remote communication is allowed). However, the language also provides a (high-level) primitive for remote communication, that transparently delivers a message to an agent even if the latter is not co-located with the sender. This primitive is then encoded in the low-level calculus by a central forwarding server, implemented by only using the low-level primitives.

Confined- λ [98]. *Confined- λ* is a higher-order functional language that supports distributed computing by allowing expressions at different localities to remotely communicate via located channels. *Confined- λ* is a typed language: the transmissible process abstractions can be parameterised with respect to channel names, and the types of transmissible values permit restricting the subsystem where a value can freely move (for more comments on the type system, see Chapter 4).

Ambient-like languages

The Ambient Calculus [41]. The Ambient calculus is an elegant notation to model hierarchically structured distributed applications. The calculus is centred around the notion of connections between ambients, that are at the same time administrative domains and computational environments. The focus of the calculus is on the mobility primitives, for entering, exiting and dissolving ambients. Each primitive can be executed only if the ambient hierarchy is structured in a precise way; e.g., an ambient n can enter an ambient m only if n and m are sibling, i.e. they are both contained in the same ambient. Inter-processes communication happens inside an ambient; it is asynchronous and anonymous (i.e. no named communication medium is used).

Remarkably, as shown in [41], the primitives *in*, *out* and *open* are enough to achieve Turing completeness: Turing machines and arithmetic can be coded in a direct way. These primitives can also encode communication, but explicit operations for sending and receiving messages are added for the sake of programmability.

Safe Ambients [105]. Even if elegant and concise, the Ambient calculus suffers from the lack of a rich equational theory. This is mainly caused by the so called grave interferences. Roughly speaking, these are cases where the inherent non-determinism of ambient movement goes wild. More precisely, an ambient system is said interferent whenever it contains a sub-term that can reduce in two or more ways that are logically different. Hence, it turns out that it is extremely difficult to write programs which preserve their behaviour in all contexts and so the algebraic theory of the calculus is poor.

In [105], several examples of interfering systems are provided. The analysis of their behaviour is then exploited to motivate the introduction of *co-capabilities*. In this way, the execution of each action must be authorised by the target ambient; for example, if n wants to enter m (by exercising the capability in_m), then m has to accept it (by exercising the co-capability $\overline{in_m}$). Then, each movement

is executed as the effect of a synchronisation between a capability and the corresponding co-capability. In this way, grave interferences can be better controlled and the behavioural theory of the calculus gets richer.

[107] enhances the previous framework by controlling the execution of an action by also exploiting passwords. Now, to execute an action, it is both required that the target ambient enables the action (via a co-capability) and that the exercising ambient provides a legal password to perform the action. This is a mean to achieve a sound and complete bisimulation, as further discussed in Chapter 5.

Boxed Ambients [25]. A major problem in the Ambient calculus is the *open* primitive. Indeed, by exercising it, the executing ambient embodies all the content of the dissolved ambient, including its capabilities and migration strategies. Of course, there is nothing wrong with that but it must be used very carefully. Unluckily, in Ambient its use is quite common, since it is the only mean to enable the communication between processes located in different ambients.

The key observation leading to Boxed Ambient [25] is that the expressiveness of the paradigm and the underlying philosophy are maintained by allowing a very constrained form of remote communication. Indeed, apart from local exchanges, the calculus enables the execution of input/output actions towards the father (i.e., the enclosing ambient) or the children (i.e., the enclosed ambients). Thus, the *open* primitive is dropped and directed communications replace it.

However, the key design principles of Boxed Ambients introduce several forms of non-local communication interferences (similar to the grave interferences of [105]). Thus, in [28] co-capabilities and passwords (akin to [107]) are added. The behavioural semantics of the calculus is simplified, even if its syntax and operational semantics are slightly complicated.

The Seal Calculus [45]. Similarly to Boxed Ambients, the Seal calculus is a variant of Ambient without the *open* primitive and with the possibility of having parent/child synchronous communication. Differently from all the previous variants of Ambient, communication is channel-based (*à la* π -calculus). Moreover, whole ambients can be passed through a channel. This is the only way to program ambient movements: indeed, no primitive such as *in* or *out* is present. Thus, an ambient can move from n to m if it is sent along a channel by a process within n and is received by a process within m that puts it in execution. Hence, the calculus is similar to Safe Ambients, in that movement capabilities (i.e. output actions containing an ambient) can be reduced only if appropriate co-capabilities (input actions over the same channel) are present in the receiving ambient.

Mobile Resources [77]. The calculus is a CCS-derived language whose general structure has been inspired by Ambient: resources are contained within a named slot (the equivalent of an ambient) and, since resources can be slot themselves, there is a hierarchical nesting structure. Resources can be moved within the hierarchy and their movement crosses slot boundaries with the assurance that no resource can be created (i.e. capacity constraints are respected).

Chapter 3

Types to Control Process Activities

As we said in the Introduction, code mobility is a fundamental aspect of global computing; however it gives rise to a lot of relevant security problems, other than attacks to inter-process communication over the communication channels (e.g. traffic analysis, message modifications/forging). For instance, malicious mobile processes can attempt to perform illegal actions while running in the nodes hosting them. Hence, a server receiving a mobile process for execution needs to impose strong requirements to ensure that the incoming process does not violate the access control policy it imposes on its code. Such problems have increasingly importance due to the spreading of security critical applications, like, e.g., electronic commerce and on-line bank transactions.

Several alternative approaches have been exploited to enforce access control policies in distributed computing systems, ranging from type systems [59, 38, 90] to control and data flow analysis [120, 67, 89], from abstract interpretation [83, 104] to proof carrying code [117]. The approaches may differ in the level of trust required, the flexibility of the enforced policy and their costs to components producers and users. Some sensible and flexible language-based security techniques are investigated in [70].

In this chapter, we present the type theory we developed to control resource accesses and process mobility in μKLAIM . We start by intuitively illustrating our approach and by providing some key principles that drew our works (Section 3.1). Then, in Section 3.2, we present a basic capability-based type system that meets our requirements. Sections 3.3 and 3.4 contain two (orthogonal) evolutions of the basic system: the first one shows how our theory can be tailored to allow dynamic management of capabilities, the second one refines our types to express finer-grained policies. Finally, Section 3.5 reviews some related papers appeared in literature on this subject, and contrasts our theory against them.

3.1 Overview: the Approach and Key Principles

The idea of statically controlling the execution of a program via types dates back in time. However, to better deal with global computing problems, we generalised traditional types to types describing process behaviours. Intuitively, these types provide information about the intentions of processes, namely the operations processes can perform at a specific locality (downloading/consuming tuples, producing tuples, activating processes and creating new nodes). By using types, each node comes equipped with a policy, specified in terms of execution capabilities: the policy of node l describes the actions processes located at l are allowed to execute. Type checking will guarantee that only processes whose intentions match the rights provided by the node executing them are allowed to proceed.

Capabilities and Types

To formalise the intuitions we have just outlined, we first define capabilities and types.

Definition 3.1.1 (Capabilities and Types) *The set of capabilities C is $\{r, i, o, e, n\}$. We let Π be the powerset of C and use π to range over Π .*

Types, ranged over by δ , are functions mapping \mathcal{L} into Π such that $\delta(l) \neq \emptyset$ only for finitely many l s.

For the sake of readability, a type mapping l_i to a non-empty π_i , for $i = 1, \dots, k$, will be written as $[l_i \mapsto \pi_i]_{i=1, \dots, k}$. Intuitively, r, i, o, e and n indicate the operation whose name begins with it. For example, e is used to control process mobility; thus, the capability $[l' \mapsto \{e\}]$ in the type of locality l will enable processes running at l to perform **eval** actions over l' . Notice that, once spawned from l to l' , a process can take advantage of the capabilities offered by l' and will not use anymore capabilities offered by l . This models real-life scenarios and supports our view of policies as descriptions of *local* behaviours. Thus, if l does not allow inputs from l'' while l' does, then the agent **eval**(**in**(\dots)@ l'')@ l' is perfectly legal at l .

To simplify notation, we permit that $n \in \delta(l')$ also if δ is the type of a node $l \neq l'$; indeed, the fact that $n \in \delta(l')$ or not is of no importance because the syntax prescribes to execute action **new** always locally.

We now introduce an ordering relation over types to formalise degrees of restrictions among policies of nodes. To this aim, we start with defining an ordering over sets of capabilities that will induce the desired ordering on types.

Definition 3.1.2 $\pi_1 \sqsubseteq_{\Pi} \pi_2$ if and only if $\pi_2 \subseteq \pi_1$.

Thus, if $\pi_1 \sqsubseteq_{\Pi} \pi_2$ then π_1 enables at least the actions enabled by π_2 . Here, the ordering \sqsubseteq_{Π} is borrowed from [80]. However, the type theory we develop is completely parametric with respect to the used ordering over capabilities and other alternatives are possible (see, e.g., [59]).

By taking advantage of the fact that types are functions, we express *subtyping* in terms of the standard pointwise inclusion of functions.

Definition 3.1.3 (Subtyping) *We say that δ_1 is a subtype of δ_2 (or that δ_2 is a supertype of δ_1), written $\delta_1 \leq \delta_2$, if $\delta_2(l) \sqsubseteq_{\Pi} \delta_1(l)$ for every $l \in \mathcal{L}$.*

Subtyping formalises the idea that, if $\delta_1 \leq \delta_2$, then δ_1 expresses a less permissive policy than δ_2 . Notice that subtyping is easily decidable because we need to check $\delta_2(l) \sqsubseteq_{\Pi} \delta_1(l)$ only for those l such that $\delta_1(l) \neq \emptyset$.

Typed syntax

The syntax of μKLAIM , as reported in Table 2.4, must be slightly modified to introduce typing information in the syntax. First, nodes must be equipped with a type describing their policy and become triples of the form

$$l ::^{\delta} C$$

Moreover, dynamically created nodes (via the action **new**) must also be associated with a policy. We can decide to assign them some kind of ‘default policy’ (e.g., the policy of their creators) but, for the sake of flexibility, we prefer to explicitly specify a δ as argument of actions **new**, that now become

$$\mathbf{new}(l : \delta)$$

Finally, we also find it convenient to require an explicit declaration of how a name received via an action **read/in** is used by the continuation process. Thus, names bound by templates are now annotated with a set of capabilities π and take the form

$$!x : \pi$$

Intuitively, in **read**(! $x : \pi$)@ $l.P$ process P only needs the capabilities $[x \mapsto \pi]$ to be executed, i.e. π specifies the access rights corresponding to the operations that P wants to perform over (the name that will replace) x .¹

Our Approach

In order to justify our typing approach, we first sketch how a completely static type system could be developed. We shall see, however, that this approach violates some principles of global computing and of tuple spaces; this will lead us to a more suitable typing approach.

First, for every action of the form **eval**(P)@ l in the net, we have to verify that P complies with δ , where δ is the type associated to the node with address l in the net.

¹Notice that the π associated to x is not strictly necessary: it can be inferred by examining how the continuation process P uses x . However, its presence enables a simpler static type checking. An example where a typing information is inferred from the continuation process will be given in Section 4.2; a similar approach can be used here.

This implies that a full knowledge of the net is needed during type checking. The problem here is that, in the setting of global computers, one would desire to have checkings that could be performed locally, i.e. that could be carried on without any remote information.

The case for actions **out**, **read** and **in** is even more problematic. Indeed, we have to statically control the exchanges that can take place within a net. By following the approach of [126] for the π -calculus, we could associate to the tuple space of a node l a *sorting* that constraints the kind of tuples the node can host (i.e., the tuple space only hosts tuples ‘of the same shape’). While this can be acceptable for a channel, that traditionally represents a port or a method invocation, this is quite unrealistic for a tuple space. Indeed, tuple spaces usually model large portions of memory, that can unlikely store data of the same kind.

We now sketch the solutions we propose to overcome the problems presented above. Actions **eval** are typed akin to [88], by deferring at runtime the typing of spawned agents. Thus, when l tries to spawn a process P to l' (whose policy is δ'), we require that

$$\delta' \vdash_{\nu} P$$

This judgement intuitively means that P can run at l' without violating δ' .

Compliance between the typing information in a tuple and that in a template is checked when executing the corresponding actions **in** and **read**; thus, no sorting is assigned to tuple spaces. This does not make the semantics too inefficient because (RED-IN)/(RED-READ) already invoke function *match*. Thus, we only need to charge *match* with the burden of verifying type compliance between the accessed tuple, t , and the template T used to access it.

3.2 A Basic Type System

We now formalise the intuitive ideas presented in the last section. As we already said, each node is decorated with a type that determines the access policy of the node in terms of access rights on the other nodes of the net. We show the mixture of static and dynamic checks needed to ensure the controls on resource accesses and processes mobility we aim at, and we prove soundness of our solution.

3.2.1 Static Semantics

A static type checker verifies whether the processes in the net do comply with the access control policies of the nodes where they are allocated. Thus, for each node of a net, say $l ::^{\delta} C$, the static type checker procedure can determine if the actions that C intends to perform when running at l are enabled by the access policy δ or not. Moreover, the type checker verifies that the declarations made for localities bound by actions **in** and **read** are consistent with the way in which the continuation process uses them.

$\frac{}{\Gamma \vdash_l \mathbf{nil}}$	$\frac{}{\Gamma \vdash_l \langle t \rangle}$	$\frac{}{\Gamma \vdash_l X}$
$\frac{\Gamma \vdash_l P}{\Gamma \vdash_l \mathbf{rec} X.P}$		$\frac{\Gamma \vdash_l C_1 \quad \Gamma \vdash_l C_2}{\Gamma \vdash_l C_1 \mid C_2}$
$\frac{\Gamma(u) \sqsubseteq_{\Pi} \{o\} \quad \Gamma \vdash_l P}{\Gamma \vdash_l \mathbf{out}(t)@u.P}$		$\frac{\Gamma(u) \sqsubseteq_{\Pi} \{e\} \quad \Gamma \vdash_l P}{\Gamma \vdash_l \mathbf{eval}(Q)@u.P}$
$\frac{\Gamma(u) \sqsubseteq_{\Pi} \{i\} \quad \mathit{upd}(\Gamma, T) \vdash_l P}{\Gamma \vdash_l \mathbf{in}(T)@u.P}$		$\frac{\Gamma(u) \sqsubseteq_{\Pi} \{r\} \quad \mathit{upd}(\Gamma, T) \vdash_l P}{\Gamma \vdash_l \mathbf{read}(T)@u.P}$
$\frac{\Gamma(l) \sqsubseteq_{\Pi} \{n\} \quad \delta \leq \Gamma \cup [l' \mapsto \Gamma(l)] \quad \Gamma \cup [l' \mapsto \Gamma(l)] \vdash_l P}{\Gamma \vdash_l \mathbf{new}(l' : \delta).P}$		

Table 3.1: Static Inference Rules for the Basic Type System

The type judgement takes the form $\Gamma \vdash_l C$, where Γ is called *type context* and it is indeed a type. It is used to keep track of the local policy of the node where the inference takes place and to record the type annotations specified within a template T when typing the continuation of an action **in** or **read**. The updating of a type context with the type annotations specified within a template T is denoted $\mathit{upd}(\Gamma, T)$ and is formally defined as

$$\mathit{upd}(\Gamma, T) = \begin{cases} \mathit{upd}(\mathit{upd}(\Gamma, T_1), T_2) & \text{if } T = T_1, T_2 \\ \Gamma \cup [x \mapsto \pi] & \text{if } T = !x : \pi, \\ \Gamma & \text{otherwise} \end{cases}$$

where \cup denotes the pointwise union of functions.

We now comment on the type checking rules of Table 3.1. Rules (TB-NIL), (TB-DATUM) and (TB-VAR) state that process **nil**, datum $\langle t \rangle$ and process variable X do not affect the behaviour of a component located in a node. Rule (TB-REC) type checks a recursive process by type checking its body, while rule (TB-PAR) type checks in isolation parallel components located at the same node. The main rules are those for action prefixing. In all these rules, the static checker verifies the existence of the capability for executing the checked action in the current typing environment. In rule (TB-OUT), this is the only check performed, together with the fact that the continuation type checks. A similar situation arises in rule (TB-EVAL); indeed, in general nothing can be statically said about the legacy of Q at u . Since u can be instantiated at runtime (upon execution of a **in/read** binding u and prefixing

the **eval** action), the locality name replacing u (and hence its associated policy) will be known only at run-time. In rules (TB-IN) and (TB-READ), the continuation process P can intend to perform actions on the names bound by T , as specified by the capabilities associated to u in T . Thus, P must be typed in the environment obtained from Γ by adding such information on these names. In rule (TB-NEW), we assume that the creating node owns over the created one all the capabilities it owns on itself²; thus, the continuation process P will be typed in the environment Γ extended with the association $[l' \mapsto \Gamma(l)]$. Moreover, the check $\delta \leq \Gamma \cup [l' \mapsto \Gamma(l)]$ verifies that the access policy δ for the new node is in agreement with (i.e., less permissive than) the policy Γ of the node executing the operation. This check prevents a malicious node l from forging capabilities by creating a new node with more powerful capabilities (where, e.g., sending a malicious process that takes advantage of capabilities not owned by l).

Remark 3.2.1 Notice that the type checker does not verify whether each process invocation (via a process variable X) is always associated with a recursive process declaration (i.e., a binder **rec** X). Indeed, this does not compromise the security we aim at: an undefined variable turns out to be stuck, since in the operational semantics no rule is given for it. Similarly, the type checker does not verify whether the bound names occurring in a process are all distinct; we assume they are. Control of both these features can be integrated in a standard way in our type system; we omit it for the sake of simplicity.

To conclude, we define well-typed nets as nets where each node can successfully pass a static type checking phase.

Definition 3.2.2 (Well-typedness) *A net is well-typed if, for each (restricted or not restricted) node $l ::^\delta C$, it holds that $\delta \vdash_l C$.*

3.2.2 Dynamic Semantics

As we have already illustrated, the operational semantics of Table 2.5 needs to be modified by adding some runtime type checks. Since each node comes equipped with one policy (that can be modified through computations, see later on), we want to impose that all clones of the same node have the same policy. The simplest way to enforce this requirement is to only deal with *well-formed nets*, i.e. nets where distinct nodes have pairwise different addresses; here and in the next chapter we only consider well-formed nets. Clearly, well-formedness has to be preserved by the operational semantics. Unfortunately, by inspecting the rules in Tables 2.2 and 2.5, it turns out that this is not the case: by using rules (RED-STRUCT) and

²It is necessary to modify the policy of the creating node to also allow operations over the created one, otherwise the new node would be useless. Indeed, no other node of the net could perform any operation on it (since it is fresh, its name cannot be in the policy of any other node) and it could not perform any operation (since no code is located on it). Our solution was driven by the sake of simplicity.

$match_\delta(l', l') = \epsilon$	$\frac{\delta(l') \sqsubseteq_{\Pi} \pi}{match_\delta(!x : \pi, l') = [l'/x]}$
$match_\delta(T_1, t_1) = \sigma_1$	$match_\delta(T_2, t_2) = \sigma_2$
$match_\delta(T_1, T_2, t_1, t_2) = \sigma_1 \circ \sigma_2$	

Table 3.2: Typed Matching Rules

(STR-CLONE) it is possible to turn a well-formed net into an ill-formed one. To overcome this problem, we remove (STR-CLONE) from the definition of structural congruence. However, notice that the main reduction rules can only be used when there is exactly one component located at each node; rule (STR-CLONE), possibly together with rule (STR-ABS), ensured that this requirement was always satisfiable, also when the action is performed locally. Hence, we need to properly adapt the operational rules to cope with this deficiency. Our solution is to add a new reduction rule, (RED-SPLIT), that permits splitting on-the-fly the parallel components running at a node into clones, thus enabling the application of the main reduction rules. This *temporarily* turns a well-formed net into an ill-formed one; however, we shall prove that the reduction relation still relates well-formed nets only. In conclusion, (RED-SPLIT) permits a compact and general formulation of the reduction rules without the need of explicitly considering all the parallel components running at a node and of having different rules for local and remote operations.

We are now ready to give the dynamic semantics of typed μ KLAIM. The structural congruence is readily adapted from Table 2.2 (remember that in μ KLAIM the allocation environments ρ are all empty and, thus, omitted); we only need to set rules (STR-ABS) and (STR-REC) to be

$$l ::^\delta C \equiv l ::^\delta C \mid \mathbf{nil} \qquad l ::^\delta \mathbf{rec} X.P \equiv l ::^\delta P[\mathbf{rec} X.P/X]$$

and to ignore rule (STR-CLONE).

Pattern-matching must now take into account typing information. Its formal definition is given in Table 3.2, while the intuitions behind it have already been given in Section 3.1. The main novelty is that, when l tries to replace a template field $!x$ declared at π with the name l' , the capabilities owned by l over l' must be at least (w.r.t. the ordering \sqsubseteq_{Π}) π .

Finally, the μ KLAIM operational semantics with types for resource access and process mobility control is the least relation induced by the rules in Table 3.3. The key points w.r.t. the relation defined in Table 2.5 are the premise of rule (RED-EVAL) and the introduction of the typed version of function *match* in rules (RED-IN) and (RED-READ). Also notice that in rule (RED-NEW) the type of the creating node is extended to collect capabilities over the newly created node. All these features have been previously motivated; thus, we omit any further comment here.

We end this section by proving that the reduction relation preserves nets' well-formedness. As usual, we shall write \mapsto^* to denote the reflexive and transitive closure of \mapsto .

$\frac{\text{(RED-OUT)}}{l ::^\delta \mathbf{out}(t)@l'.P \parallel l' ::^{\delta'} \mathbf{nil} \mapsto l ::^\delta P \parallel l' ::^{\delta'} \langle t \rangle}$
$\frac{\text{(RED-EVAL)} \quad \delta' \vdash_{l'} Q}{l ::^\delta \mathbf{eval}(Q)@l'.P \parallel l' ::^{\delta'} \mathbf{nil} \mapsto l ::^\delta P \parallel l' ::^{\delta'} Q}$
$\frac{\text{(RED-IN)} \quad \mathit{match}_\delta(T, t) = \sigma}{l ::^\delta \mathbf{in}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^\delta P\sigma \parallel l' ::^{\delta'} \mathbf{nil}}$
$\frac{\text{(RED-READ)} \quad \mathit{match}_\delta(T, t) = \sigma}{l ::^\delta \mathbf{read}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^\delta P\sigma \parallel l' ::^{\delta'} \langle t \rangle}$
$\frac{\text{(RED-NEW)}}{l ::^\delta \mathbf{new}(l' : \delta').P \mapsto (\nu l')(l ::^\delta \uplus [l' \mapsto \delta(l)] P \parallel l' ::^{\delta'} \mathbf{nil})}$
$\frac{\text{(RED-SPLIT)} \quad l ::^\delta C_1 \parallel l ::^\delta C_2 \parallel N \mapsto (\nu \bar{l})(l ::^{\delta'} C'_1 \parallel l ::^\delta C'_2 \parallel N')}{l ::^\delta C_1 C_2 \parallel N \mapsto (\nu \bar{l})(l ::^{\delta'} C'_1 C'_2 \parallel N')}$
<p>with rules (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 2.3</p>

Table 3.3: Typed Operational Semantics for μKLAIM

Proposition 3.2.3 *If N is well-formed and $N \mapsto^* N'$, then N' is well-formed.*

Proof: It is easy to prove, by induction on the rules, that the (modified) structural congruence preserves well-formedness of nets. Thus, we are only left to prove that the reduction relation does never transform a well-formed net into a net where two distinct nodes have the same address (indeed, the reduction rules could also be applied to nets that do not satisfy this property). To this aim, we first prove a Lemma stating that a single reduction step from a net N preserves the number of nodes having the same address. This property is expressed by using $\mathit{clone}(N)$ to denote the least number of nodes that should be removed from N in order the remaining nodes have different addresses (i.e. the remaining net to be well-formed). To formally define function $\mathit{clone}(\cdot)$, we exploit the auxiliary function $\mathit{mnl}(\cdot)$ (mnl stands for ‘multiset of node localities’), that when applied to a net returns the multiset of localities naming the nodes of the net, and is inductively defined over the syntax of nets as follows:

$$\mathit{mnl}(l ::^\delta C) = \{\!\!\{ l \}\!\!\} \quad \mathit{mnl}(N_1 \parallel N_2) = \mathit{mnl}(N_1) \amalg \mathit{mnl}(N_2)$$

where $\{\!\!\{ l_0, \dots, l_n \}\!\!\}$ denotes the multiset with elements l_0, \dots, l_n and \amalg denotes mul-

tiset union. Now, for any μKLAIM net N , we can define $\text{clone}(N)$ as the cardinality of the multiset obtained by removing from $\text{mnl}(N)$ one occurrence of each different locality occurring in it.

Lemma 3.2.4 *If $N \mapsto N'$ then $\text{clone}(N) = \text{clone}(N')$.*

Proof: We reason by induction on the length of the proof of the reduction $N \mapsto N'$. The base case (i.e. one of the axioms (LTS-OUT), (LTS-EVAL), (LTS-IN), (LTS-READ), (LTS-NEW) has been used) is obvious. In the inductive case, we reason by case analysis on the last rule applied. The cases of rules (LTS-PAR), (RED-RES) and (LTS-STRUCT) (it can be easily seen that \equiv preserves $\text{clone}(\cdot)$) easily follow by induction. Suppose now that the last applied rule is (RED-SPLIT) and let $N_1 \mapsto N_2$ be its premise. Then, due to the form of the nets involved in the rule, we have $\text{clone}(N_1) = \text{clone}(N) + 1$ and $\text{clone}(N_2) = \text{clone}(N') + 1$. Since the proof of $N_1 \mapsto N_2$ is shorter than that of $N \mapsto N'$, we can apply induction and deduce that $\text{clone}(N_1) = \text{clone}(N_2)$, from which it follows that $\text{clone}(N) = \text{clone}(N')$ that proves the thesis.

To conclude, note that a net N is well-formed if, and only if, $\text{clone}(N) = 0$. Hence, by using Lemma 3.2.4 and by a straightforward induction on the length of reduction sequences, the thesis easily follows. ■

3.2.3 Type Soundness

We first prove some standard results for type systems, i.e. weakening and substitutivity. Then, we show that (the modified) structural congruence preserves well-typedness; all these results will enable us to prove that the reduction relation preserves well-typedness. Then, we formalise the runtime errors we aim at eliminating and prove that in well-typed nets runtime errors never occur. Finally, we can state that a well-typed net never gives rise to runtime errors along any computation. This result will be also generalised to deal with only partially well-typed nets.

Lemma 3.2.5 (Weakening) *If $\Gamma \vdash_l C$ then $\Gamma' \vdash_l C$, for every Γ' such that $\Gamma \leq \Gamma'$.*

Proof: Trivial. ■

Lemma 3.2.6 (Substitutivity)

1. *If $\Gamma \vdash_l C$ then, for any substitution σ , $\Gamma\sigma \vdash_l C\sigma$.*
2. *If $\Gamma \vdash_l C$ and $\Gamma \vdash_l Q$, then $\Gamma \vdash_l C[Q/X]$.*

Proof:

1. The proof is by induction on length of the inference of the type judgement. The base cases (i.e., rules (TB-NIL), (TB-DATUM) and (TB-VAR)) are trivial. Let us examine the case in which the last rule used is (TB-IN); the cases for (TB-READ), (TB-OUT), (TB-EVAL), (TB-NEW) and (TB-PAR) are similar or

easier. By definition, $C = \mathbf{in}(T)@l'.P$, $\Gamma(l') \sqsubseteq_{\Pi} \{i\}$ and $\text{upd}(\Gamma, T) \vdash_l P$. Without loss of generality, we can assume that $\text{dom}(\sigma) \cap \text{bv}(T) = \emptyset$ (otherwise, if this is not the case, we could rename the bound names); thus we have that $C\sigma = \mathbf{in}(T\sigma)@(l'\sigma).(P\sigma)$. Now, by induction, we have that $(\text{upd}(\Gamma, T))\sigma \vdash_l P\sigma$ that easily implies $\text{upd}(\Gamma\sigma, T\sigma) \vdash_l P\sigma$. Finally, it is clear that $(\Gamma\sigma)(l'\sigma) \sqsubseteq_{\Pi} \{i\}$; thus, we can apply rule (TB-IN) and conclude.

2. The proof proceeds by induction on length of the inference of the type judgement. The base cases, i.e. rules (TB-NIL), (TB-DATUM) and (TB-VAR), are trivial. If the last used rule is (TB-REC), the thesis easily follows by induction (recall that substitution application may require renaming of bound variables to avoid capturing free names). If the last used rule is (TB-PAR), the thesis follows by induction using the fact that $(C_1|C_2)\sigma = C_1\sigma|C_2\sigma$. The cases when the last rule used is (TB-IN), (TB-READ), (TB-OUT), (TB-EVAL) and (TB-NEW) are similar; we examine only the first one. By definition, $C = \mathbf{in}(T)@l'.P$, $\Gamma(l') \sqsubseteq_{\Pi} \{i\}$ and $\text{upd}(\Gamma, T) \vdash_l P$. By hypothesis, $\Gamma \vdash_l Q$; thus, by using Lemma 3.2.5, we have that $\text{upd}(\Gamma, T) \vdash_l Q$, since we can always assume that $\text{bv}(T) \cap \text{fv}(Q) = \emptyset$. Then, by using the induction hypothesis, we get that $\text{upd}(\Gamma, T) \vdash_l P[Q/X]$ and, by applying rule (TB-IN), we conclude. ■

Lemma 3.2.7 *If N is well-typed and $N \equiv N'$, then N' is well-typed.*

Proof: By mutual induction on the length of the inferences for $N \equiv N'$ and $N' \equiv N$. The base case covers the axioms in Table 2.2, except rule (STR-CLONE), and reflexivity of \equiv . All the cases are trivial; just notice that the static typing is not affected if we coherently rename bound names within a net and that rule (STR-REC) relies on Lemma 3.2.6(2). The inductive step (for symmetry, transitivity and context closure) are easy. ■

Theorem 3.2.8 (Subject Reduction) *If N is well-typed and $N \mapsto N'$, then N' is well-typed.*

Proof: By induction on the length of the inference of $N \mapsto N'$.

Base Step: We reason by case analysis on the axioms (i.e. the first five rules) of Table 3.3.

(LTS-OUT). Since by hypothesis N is well-typed, we have that $\delta \vdash_l \mathbf{out}(t)@l'.P$. Due to the form of the process involved, rule (TB-OUT) has been the last one applied to deduce the type judgement; hence we also have that $\delta \vdash_l P$. Moreover, by applying (TB-PAR) to $\delta' \vdash_{l'} C'$ (that holds by hypothesis) and to $\delta' \vdash_{l'} \langle t \rangle$ (axiom (TB-DATUM)), we get that $\delta' \vdash_{l'} C' | \langle t \rangle$. This suffices to conclude that N' is well-typed.

(LTS-EVAL). By reasoning like before, we can prove that $\delta' \vdash_{l'} P' | Q$. This easily follows by applying (TB-PAR) to $\delta' \vdash_{l'} P'$, that holds by hypothesis, and to $\delta' \vdash_{l'} Q$, that is the premise of rule (RED-EVAL).

(LTS-IN). Since it holds that $\delta' \vdash_{l'} \mathbf{nil}$, we are only left to prove that $\delta \vdash_l P\sigma$. Now, by hypothesis, we have that $\delta \vdash_l \mathbf{in}(T)@l'.P$, where rule (TB-IN) has been the last one applied to infer the judgement; hence we also have that $\text{upd}(\delta, T) \vdash_l P$. Now, let $\{x_i : \pi_i\}_{i \in I}$ be the formal fields of T and $\{l_i\}_{i \in I}$ be the corresponding names of t . By the premise of rule (LTS-IN) and by the definition of match^δ from Table 3.2, we have that $\text{match}_\delta(T, t) = \sigma$, where $\sigma = [l_i/x_i]_{i \in I}$, and $\delta(l_i) \sqsubseteq_{\Pi} \pi_i$ for every $i \in I$. This means that $[l_i \mapsto \pi_i]_{i \in I} \leq \delta$ and $\text{upd}(\delta, T) = \delta \cup [x_i \mapsto \pi_i]_{i \in I}$; thus, $(\text{upd}(\delta, T))\sigma = \delta$. By Lemma 3.2.6(1), we can conclude.

(LTS-READ). Similar to the previous case.

(LTS-NEW). By hypothesis we have that $\delta \vdash_l \mathbf{new}(l' : \delta').P$, where rule (TB-NEW) has been the last one applied to infer the judgement. Hence we also have that $\delta \cup [l' \mapsto \delta(l)] \vdash_l P$. The thesis follows by using Lemma 3.2.6(1) with substitution $\sigma = [l'/l]$.

Inductive Step: We reason by case analysis on the last applied operational rule of Table 3.3.

(RED-SPLIT). By hypothesis, we have that $\delta \vdash_l C_1|C_2$. Due to the form of the process involved in the judgement, rule (TB-PAR) has been the last one applied to deduce the judgement; hence we also have that $\delta \vdash_l C_1$ and $\delta \vdash_l C_2$. Thus, we have that the net $l ::^\delta C_1 \parallel l ::^\delta C_2 \parallel N$ is well-typed. By induction, we get that $(\nu \bar{l})(l ::^{\delta'} C_1 \parallel l ::^{\delta'} C_2 \parallel N')$ is well-typed. It is easy to prove that $\delta(u) \subseteq \delta'(u)$ for each $u \in \mathcal{L}$; thus, the thesis directly follows by using Lemma 3.2.5.

(RED-PAR). By hypothesis, $N_1 \parallel N_2$ is well-typed, hence N_1 and N_2 are well-typed too. Now, by induction, N'_1 is well-typed and hence $N'_1 \parallel N_2$ is well-typed.

(RED-RES). By definition, $(\nu l)N$ is well-typed if and only if N is well-typed; this suffices to conclude by a straightforward induction.

(RED-STRUCT). From the hypothesis, N is well-typed and $N \equiv N_1$; by Lemma 3.2.7, it follows that N_1 is well-typed too. Now, by induction, we get that N_2 is well-typed. From this fact and from the hypothesis $N_2 \equiv N'$, again by Lemma 3.2.7, it follows that N' is well-typed. ■

Now, we introduce the notion of run-time error and state type safety, i.e. that well-typed nets do not give rise to run-time errors. *Run-time errors* are defined by the rules in Table 3.4 in terms of predicate $N \nearrow$ that holds true when, within the net N , a process P located at a node with address l attempts to perform an action that is not allowed by the access policy of the node. The rules are straightforward.

Notation 3.2.9 Here and in what follows, given an action a different from **new**, we use $\text{arg}(a)$ to denote its argument, $\text{tgt}(a)$ its target location and $\text{cap}(a)$ the capability

$\text{(ERRACT)} \frac{\delta(\text{tgt}(a)) \not\sqsubseteq_{\Pi} \{\text{cap}(a)\}}{l ::^{\delta} a.P \nearrow}$	$\text{(ERRPAR)} \frac{N \nearrow}{N \parallel N' \nearrow}$
$\text{(ERRRES)} \frac{N \nearrow}{(\nu l)N \nearrow}$	$\text{(ERRSTR)} \frac{N \equiv N' \quad N' \nearrow}{N \nearrow}$

Table 3.4: Run-Time Error

corresponding to a . For example, if a is $\mathbf{out}(t)@l$, then we have $\text{arg}(a) = t$, $\text{tgt}(a) = l$ and $\text{cap}(a) = o$.

Theorem 3.2.10 (Type Safety) *If N is well-typed then $N \nearrow$ does not hold.*

Proof: We proceed by contradiction and prove, by induction on the length of the inference of $N \nearrow$, that, if $N \nearrow$, then N is not well-typed.

Base Step: In this case, the error is generated by using axiom (ERRACT). This means that N is a node of the form $l ::^{\delta} a.P$ and $\delta(\text{tgt}(a)) \not\sqsubseteq_{\Pi} \{\text{cap}(a)\}$. Therefore, node $l ::^{\delta} a.P$, and hence N , is not well-typed otherwise rules (TB-OUT), (TB-EVAL), (TB-IN), (TB-READ) or (TB-NEW) would have failed.

Inductive Step: By case analysis on the last error rule used.

(ERRPAR). From the premise $N \nearrow$ of the rule and by induction, we have that N is not well-typed. Hence, by definition, $N \parallel N'$ is not well-typed too.

(ERRRES). Similar.

(ERRSTR). From the premise $N' \nearrow$ of the rule and by induction, we have that N' is not well-typed. Then the thesis follows from the premise $N \equiv N'$, by using Lemma 3.2.7. ■

Therefore, well-typed nets cannot immediately give rise to run-time errors. Now, by combining together the results shown so far, we get that well-typed nets never generate run-time errors along sequences of reductions.

Corollary 3.2.11 (Global Type Soundness) *If N is well-typed and $N \mapsto^* N'$, then $N' \nearrow$ does not hold.*

Proof: The proof proceeds by induction on the length of $N \mapsto^* N'$. The base step is Theorem 3.2.10, while the inductive step follows from Theorems 3.2.8 and 3.2.10. ■

Type soundness is one of the main goal of any type system. However, in our framework it is formulated in terms of a property requiring the typing of whole nets. While this could be acceptable for LANs, where the number of hosts is usually relatively small, it is unreasonable for WANs, where in general hosts are under the

control of different authorities. When dealing with larger nets, it is certainly more realistic to reason in terms of parts of the whole net. Hence, we put forward a more *local* formulation of our properties and results. To this aim, we define the *restriction* of a net N to a set of localities S , written N_S , as the subnet obtained from N by deleting all nodes whose addresses are not in S . The wanted local type soundness result can be formulated as follows.

Theorem 3.2.12 (Local Type Soundness) *Let N be a net and $S \subseteq \text{fv}(N)$. If N_S is well-typed and $N \mapsto^* N'$, then $N'_S \not\rightarrow$ does not hold.*

Notice that in the previous statement, no assumption on the whole net N is made. The proof is similar to the proof of Corollary 3.2.11; in fact, local type soundness is enforced by the dynamic checking performed when processes migrate, which prevents ill-typed processes to get into N_S .

3.3 Dynamic Management of Capabilities

The basic type system presented in the previous section suffers from the fact that policies are fixed and can never change during computations. Indeed, mechanisms supporting modifications at run-time of access control policies and process capabilities turn out to be essential for dealing with pervasive network applications, like, e.g., those for e-commerce.

In this section, we discuss how our theory can be extended to take into account very flexible and sophisticated mechanisms to dynamically handle capabilities. For the sake of presentation, we will first introduce only the possibility for a policy to be enlarged with new capabilities; then, we will argue on how implementing further desirable features, like capability consumption, expiration and revocation.

Types and Grantings

As we have already said, actions **new** can only be performed locally. Hence, the corresponding capability cannot be passed through. Since in this section we focus on capability passing, we only consider capabilities i , r , o and e , and, for the sake of simplicity, we assume that actions **new** are enabled everywhere. Types are defined like in Definition 3.1.1 over this restricted set of capabilities. However, now types can also change during computations; recall that the pointwise union of functions δ_1 and δ_2 , written $\delta_1 \uplus \delta_2$, is the type δ such that $\delta(u) = \delta_1(u) \cup \delta_2(u)$ for each $u \in \mathcal{L}$.

We allow capabilities to be passed at runtime by following the *discretionary access control* model, where each principal owning a capability can pass it through. Policy modifications could be carried on by using specific primitives, like in [53]. However, for the sake of economy, we prefer to merge it in the communication primitives. This means that capabilities are associated to tuples when performing an action **out** and are retrieved when performing actions **in/read**. More precisely,

each name occurring in a tuple argument of actions **out** is now equipped with a *granting*.

Definition 3.3.1 (Capabilities and Types – revisited) *The set of capabilities C is $\{r, i, o, e\}$. We let Π be the powerset of C and use π to range over Π .*

Types, ranged over by δ , are functions mapping \mathcal{L} into Π such that $\delta(l) \neq \emptyset$ only for finitely many l s.

Definition 3.3.2 (Grantings) *Grantings, ranged over by μ , are finite partial functions mapping \mathcal{L} to Π .*

Thus, the formal syntax of tuples from Table 2.4 is now changed to be

$$t ::= u : \mu \mid t_1, t_2$$

Intuitively, grantings are used in actions **out** to point out some capabilities to be passed through together with the names in the tuple. The intended meaning is that, if l retrieves a tuple $\langle \dots, l' : [\dots, l \mapsto \pi, \dots], \dots \rangle$, then l receives the capability $[l' \mapsto \pi]$.

We could decide to let dynamically acquired capabilities be exploitable only by the process that performed the **in/read**. However, this choice would imply that the policy of the node is *not* changed. Hence, our approach consists in changing the policy of the node where the action was fired and sharing the new capabilities between all co-located processes. Moreover, since the exact moment in which an acquisition of capabilities takes place is unknown (because of the non determinism underlying the operational semantics), we let all processes already in execution at l exploit the new capabilities. This choice is driven by the sake of fairness; indeed, it seems us unfair to assign new capabilities only to newly spawned processes. Unluckily, this choice requires more runtime checks. Indeed, an action that is statically illegal could become legal upon acquisition of the capability enabling it. For this reason, if a process intends to perform an action not allowed by the policy of the node hosting it, the static typing system cannot reject the process since the capability necessary to perform the action could in principle be dynamically acquired by the node. In such cases, the typing system simply *marks* the action (turning a into \underline{a}) to require its dynamic checking. So, we shall define the operational semantics for nets written in an ‘intermediate language’, where marked actions can be used to prefix a process. Notationally, we will write \underline{P} (\underline{C} and \underline{N} , resp.) to emphasise that process P (component C and net N , resp.) may contain marked actions.

Finally, we also have to check grantings within actions **out**. This is necessary to avoid capability forging like in

$$l ::^\delta \mathbf{out}(l' : [l \mapsto \pi]) @ l. \mathbf{in}(!x : \pi) @ l$$

where $[l' \mapsto \pi] \not\leq \delta$. If the first action was legal, the second action would add new capabilities to δ and l would have enlarged its policy autonomously. This is a

clear security breach that must be avoided. Such an action **out** should be executed only if $\delta(l') \sqsubseteq_{\Pi} \pi$. However, if we perform this check statically, dynamically acquired capabilities could not be passed anymore; this somehow collides with the discretionary access control, where a dynamically received capability becomes a first-class capability (that must be handled like statically assigned ones). Thus, we defer the check on grantings at run-time too.

3.3.1 Language Semantics

In this section, we present the static and the dynamic semantics of our calculus. For the sake of presentation, we will first introduce a basic setting where capabilities can only be acquired; in Section 3.3.4 we shall enrich it to deal with further desirable features.

Static Semantics

We start with the static part of the language semantics. Informally, for each node of a net, say $l ::^{\delta} C$, the typing system determines if the actions that C intends to perform when running at l are enabled by the access policy δ or not. For example, capability e can be used to control process mobility: a process in C can migrate to l' only if $[l' \mapsto \{e\}]$ is a subtype of δ .

However, because a node can dynamically acquire capabilities when C accesses data, some actions that can be permissible at run-time could be statically illegal; such actions are marked to be dynamically checked. The marking mechanism never applies to actions whose targets are names bound by **in/read**, because such actions can be statically checked, thus alleviating the burden of dynamic checking and improving system performance. In fact, according to the syntax, whenever a name x is bound by an action **in/read**, it is annotated with a set of capabilities π that specifies the operations that the continuation process is allowed to perform by using x as the target address. Moreover, π also specifies the ‘minimal’ set of capabilities that the executing node must own or acquire over the net locality that will replace x at run-time. Hence, our type system has to reject node

$$l_1 :: [l' \mapsto \{r\}] \mathbf{read}(!x : \{o\})@l' . \mathbf{read}(!y)@x$$

because r does not belong to the annotation of x : this is a clear programming error and the type system can statically check it. On the other hand, the type system should accept node

$$l_2 :: [l' \mapsto \{r\}] \mathbf{read}(!x : \{o\})@l' . \mathbf{out}(t)@l'$$

because action **out**(t)@ l' can be marked and checked at run-time. In fact, if x is dynamically replaced with l' , l_2 will acquire capability o over l' and the process running at l_2 can proceed; otherwise, the process will be suspended. In our system, the dynamic acquisition of capabilities is exploited exactly for relaxing the

(TD-NIL) $\Gamma \vdash_l^L \mathbf{nil} \triangleright \mathbf{nil}$	(TD-DAT) $\Gamma \vdash_l^L \langle t \rangle \triangleright \langle t \rangle$
(TD-VAR) $\frac{X \in L}{\Gamma \vdash_l^L X \triangleright X}$	(TD-REC) $\frac{\Gamma \vdash_l^{L \cup \{X\}} P \triangleright \underline{P}}{\Gamma \vdash_l^L \mathbf{rec} X.P \triangleright \mathbf{rec} X.\underline{P}}$
(TD-PAR) $\frac{\Gamma \vdash_l^L C_1 \triangleright \underline{C_1} \quad \Gamma \vdash_l^L C_2 \triangleright \underline{C_2}}{\Gamma \vdash_l^L C_1 \mid C_2 \triangleright \underline{C_1} \mid \underline{C_2}}$	(TD-SND) $\frac{cap(a) \in \{o, e\} \quad \Gamma \vdash_l^L P \triangleright \underline{P}}{\Gamma \vdash_l^L a.P \triangleright \mathit{mark}_L(\Gamma)a.\underline{P}}$
(TD-RCV) $\frac{cap(a) \in \{i, r\} \quad \mathit{upd}(\Gamma, arg(a)) \vdash_l^{L \cup \mathit{bn}(arg(a))} P \triangleright \underline{P}}{\Gamma \vdash_l^L a.P \triangleright \mathit{mark}_L(\Gamma)a.\underline{P}}$	
(TD-NEW) $\frac{\delta \leq \Gamma \uplus [l' \mapsto \Gamma(l)] \quad \Gamma \uplus [l' \mapsto \Gamma(l)] \vdash_l^{L \cup \{l'\}} P \triangleright \underline{P}}{\Gamma \vdash_l \mathbf{new}(l' : \delta).P \triangleright \mathbf{new}(l' : \delta).\underline{P}}$	

Table 3.5: Typing Rules for Dynamic Capabilities Management

static type checking and admitting nodes like l_2 while requiring on (part of) them a dynamic checking.

Type judgements for processes take the form $\Gamma \vdash_l^L C \triangleright \underline{C}$. Here, L is a finite set of names and it is used to keep track of bound names that have been freed during the inference as the result of removing a binding operator, i.e. **in/read/new/rec**. The environment Γ collects together the capabilities contained in the policy of l and the type annotations for the names that have been freed in C . Intuitively, the judgement $\Gamma \vdash_l^L C \triangleright \underline{C}$ states that, when C is located at l , the unmarked actions in \underline{C} are admissible w.r.t. Γ . Instead, the marked actions in \underline{C} cannot be deemed legal at compile time but could become permissible at run-time, after dynamic acquisition of the necessary capabilities (via execution of actions **in/read** performed at l). When L is empty, we shall simply write $\Gamma \vdash_l C \triangleright \underline{C}$.

Type judgements are inferred by using the rules in Table 3.5. The function $\mathit{mark}_L(\Gamma) \cdot$ for marking process actions is defined as follows

$$\mathit{mark}_L(\Gamma)a = \begin{cases} a & \text{if } \Gamma(\mathit{tgt}(a)) \sqsubseteq_{\Pi} \{cap(a)\} \\ \underline{a} & \text{if } \Gamma(\mathit{tgt}(a)) \not\sqsubseteq_{\Pi} \{cap(a)\} \text{ and } \mathit{tgt}(a) \notin L \end{cases}$$

where $\not\sqsubseteq_{\Pi}$ denotes the negation of \sqsubseteq_{Π} . Condition $\mathit{tgt}(a) \notin L$ distinguishes actions using localities as target from those using freed names, marking the formers and rejecting the latters (as previously explained).

The rules in Table 3.5 should be quite explicative, we only remark a few points. Rule (TD-DAT) says that located tuples always successfully pass the static type

$\frac{\mu = [l_i \mapsto \pi_i]_{i=1, \dots, k} \quad \forall i = 1, \dots, k. \delta(l) \sqsubseteq_{\Pi} \pi_i}{\llbracket l : \mu \rrbracket_{\delta}} \quad \frac{\llbracket t_1 \rrbracket_{\delta} \quad \llbracket t_2 \rrbracket_{\delta}}{\llbracket t_1, t_2 \rrbracket_{\delta}}$

Table 3.6: Checking Grantings

checking, regardless their contents. This choice simplifies the technical development; however, in order to check grantings therein, we require that no tuple is present in the net at the outset (data must all be produced via actions **out**, that are dynamically checked). Rules (TD-VAR) and (TD-REC) deal with recursive definitions and only accept processes whose process variables are bound by a **rec**. Rule (TD-PAR) deals both with process composition and with component composition. Rule (TD-SND) deals with **out** and **eval**; as we already explained in Section 3.1, the arguments of these actions are not statically checked. Rule (TD-RCV) deals with actions **in** and **read**; the type annotations in the formal fields of the template are used to enrich the current type environment in order to type the continuation process. Action **new** is dealt with differently from the other actions by rule (TD-NEW) and is always statically checked (i.e. it is never marked). Indeed, **new** is always performed locally and the corresponding capability cannot be dynamically acquired.

We end this section by formalising the notion of well-typed net.

Definition 3.3.3 *A net is well-typed if, for each node $l ::^{\delta} C$, there exists a component \underline{C} such that $\delta \vdash_l C \triangleright \underline{C}$.*

Dynamic Semantics

The first ingredient we need for defining the operational semantics is a mechanism to control the capabilities passed through while executing an action **out** from node l' . This check is defined as the predicate $\llbracket \cdot \rrbracket_{\delta}$, that can be inferred by using the rules in Table 3.6. $\llbracket \cdot \rrbracket_{\delta}$ is parameterised with respect to δ , the policy of the node l' where the action **out** takes place. Intuitively, whenever a tuple passes the capabilities π_i over l to l_i (thus, the tuple is of the form $\langle \dots, l : [\dots, l_i \mapsto \pi_i, \dots], \dots \rangle$), we need to verify that l' owned such capabilities.

Another ingredient we need is a formal way to say that a template and a tuple do match. The *pattern-matching* function, $match_l^{\delta}$, is defined by the rules in Table 3.7 and is parameterised with the locality l and the access control policy δ of the node where it is invoked. A successful matching returns a type, used to extend the type of the node executing the matching with the capabilities granted by the tuple, and a substitution, used to assign names to variables in the process invoking the matching. We use σ to range over substitutions (with finite domain) of names for names, ϵ to denote the ‘empty’ substitution and \circ to denote substitutions composition. As usual, substitution application may require alpha-conversion to avoid capturing of free names.

Notice that the node where the **read/in** is executed must be authorised to access all the names occurring in the tuple accessed; this is verified by examining all the

$(M_1) \frac{l \in \text{dom}(\mu)}{\text{match}_l^\delta(l', l' : \mu) = \langle [], \epsilon \rangle}$	$(M_2) \frac{\delta(l') \cup \mu(l) \sqsubseteq_{\Pi} \pi}{\text{match}_l^\delta(!x : \pi, l' : \mu) = \langle [l' \mapsto \pi], [l'/x] \rangle}$
$(M_3) \frac{\text{match}_l^\delta(T_1, t_1) = \langle \delta_1, \sigma_1 \rangle \quad \text{match}_l^\delta(T_2, t_2) = \langle \delta_2, \sigma_2 \rangle}{\text{match}_l^\delta((T_1, T_2), (t_1, t_2)) = \langle \delta_1 \uplus \delta_2, \sigma_1 \circ \sigma_2 \rangle}$	

Table 3.7: Matching Rules for Dynamic Capabilities Management

grantings in the tuple. Indeed, the pattern-matching fails whenever l , the node where the tuple is accessed, is not named in all the grantings within the tuple. This is explicitly required in the premise of rule (M₁) and implicitly required by the fact that the $\mu(l)$ in the premise of rule (M₂) must be defined. This feature permits controlling ‘immediate access’ to tuples, i.e. constraining the nodes from where tuples can be accessed (see Section 3.3.3). Moreover, rule (M₂) ensures that if a **read/in** executed at l looks for a locality where performing the actions enabled by π , then locality l' can be selected only if the union of the capabilities over l' owned by l and of the capabilities over l' granted to l by the tuple enables π . The capabilities granted by the tuple are then used to enrich the capabilities of l over l' .

Function match_l^δ satisfies the following property, whose proof can be easily done by induction on the number of fields of the first argument of the function.

Proposition 3.3.4 *If $\text{match}_l^\delta(T, t) = \langle \delta', \sigma \rangle$ with $\text{dom}(\sigma) = \{x_i\}_{i \in I}$, then $\delta' = [l_i \mapsto \pi_i]_{i \in I}$ where, for every $i \in I$, $!x_i : \pi_i$ is a field of T , $l_i : \mu_i$ is the corresponding field of t and $\sigma(x_i) = l_i$.*

As we already said, the operational semantics relates μKLAIM nets that may contain marked actions. It is given by a reduction relation, \mapsto , which is the least relation induced by the rules in Table 3.8. Let us comment on the rules in Table 3.8. Rule (RED-OUT) says that, before adding a tuple to a TS, the grantings within the tuple must be checked according to the policy δ of the node where the process performing the **out** runs. Rule (RED-EVAL) says that a process is allowed to migrate only if it successfully passes a type checking against the access policy of the target node. During this preliminary check, some process actions could be marked to be effectively checked when at run-time. Rules (RED-IN) and (RED-READ) say that the process performing the operation can proceed only if pattern-matching succeeds. In this case, the access policy of the receiving node is enriched with the type returned by the matching mechanism and the substitution returned along with the type is applied to the continuation of the process performing the operation (and in the type annotations therein). Rule (RED-MARK) says that the in-lined security monitor stops execution whenever the capability for executing a is missing.

Notice that Proposition 3.2.3 still holds in this framework. To conclude this section, we informally present a simple but significant example where our setting turns out to be expressive and elegant; this informal presentation will be refined in Section 3.3.3. Let l_U be the address of a node representing the server of a

(RED-OUT)	$\frac{\llbracket t \rrbracket_\delta}{l ::^\delta \mathbf{out}(t)@l'.P \parallel l' ::^{\delta'} C' \mapsto l ::^\delta P \parallel l' ::^{\delta'} C' \langle t \rangle}$
(RED-EVAL)	$\frac{\delta' \vdash_l Q \triangleright \underline{Q}}{l ::^\delta \mathbf{eval}(Q)@l'.P \parallel l' ::^{\delta'} C' \mapsto l ::^\delta P \parallel l' ::^{\delta'} C' \langle \underline{Q} \rangle}$
(RED-IN)	$\frac{\mathit{match}_l^\delta(T, t) = \langle \delta'', \sigma \rangle}{l ::^\delta \mathbf{in}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^{\delta \cup \delta''} P\sigma \parallel l' ::^{\delta'} \mathbf{nil}}$
(RED-READ)	$\frac{\mathit{match}_l^\delta(T, t) = \langle \delta'', \sigma \rangle}{l ::^\delta \mathbf{read}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^{\delta \cup \delta''} P\sigma \parallel l' ::^{\delta'} \langle t \rangle}$
(RED-NEW)	$l ::^\delta \mathbf{new}(l' : \delta').P \mapsto (\nu l')(l ::^{\delta \cup [l' \mapsto \delta(l)]} P \parallel l' ::^{\delta'} \mathbf{nil})$
(RED-MARK)	$\frac{l' = \mathit{tgt}(a) \quad \delta(l') \sqsubseteq_{\Pi} \{ \mathit{cap}(a) \} \quad l ::^\delta a.P \parallel l' ::^{\delta'} C' \mapsto N}{l ::^\delta \underline{a}.P \parallel l' ::^{\delta'} C' \mapsto N}$
<p>with rule (RED-SPLIT) from Table 3.3 and rules (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 2.3</p>	

Table 3.8: Operational Semantics with Dynamic Capabilities Management

given department and let l_P be the address of a node representing the publisher of some on-line publications. We want to implement a protocol through which the department chief subscribes a ‘licence’ enabling all the department members to access the on-line publications. In terms of access control, this means that the protocol must extend the policy δ of node l_U with the capability of reading papers from l_P . Hence, upon completion of the protocol, l_U ’s policy should become $\delta \cup [l_P \mapsto \{r\}]$. Then, all the department members can spawn code over l_U and thus retrieve l_P papers by simply using the process

$$\mathbf{eval}(\mathbf{read}(\mathit{paperTitle}, !x)@l_P.\mathbf{out}(\mathit{paperTitle}, x)@l_M)@l_U$$

Action $\mathbf{read}(\mathit{paperTitle}, !x)@l_P$ looks for a paper whose title is $\mathit{paperTitle}$ and whose body is a text B ; if such a paper is found, it is copied in l_M ’s tuple space, where l_M is the address of the department member who required the paper. In a more realistic scenario, the capability ‘read’ over l_P will not be delivered forever to l_U . To model this scenario, we will also take into account means of managing capabilities loss, in Section 3.3.4.

3.3.2 Type Soundness

In this section, we first present a type soundness result that involves whole nets; then, we point out the modifications needed to get a *local* type soundness result

relative to a subnet of a larger net.

We start introducing the notion of *executable nets* that, intuitively, are nets already containing all necessary marks (as if they have already passed a static type checking phase).

Definition 3.3.5 *A net is executable if, for each node $l ::^\delta C$, it holds that $\delta \vdash_l C \triangleright C$ where, for inferring the type judgement, in addition to the rules in Table 3.5, one can also use the rules*

$$\frac{\text{(TD-MARK-SND)} \quad \text{cap}(a) \in \{o, e\} \quad \Gamma \vdash_l^L P \triangleright \underline{P}}{\Gamma \vdash_l^L \underline{a.P} \triangleright \underline{a.P}} \quad \frac{\text{(TD-MARK-RCV)} \quad \text{cap}(a) \in \{i, r\} \quad \text{upd}(\Gamma, \text{arg}(a)) \vdash_l^{L \cup \text{bn}(\text{arg}(a))} P \triangleright \underline{P}}{\Gamma \vdash_l^L \underline{a.P} \triangleright \underline{a.P}}$$

that allow a process to already contain marked actions. For the sake of readability, the judgement $\Gamma \vdash_l C \triangleright C$ will be written as $\Gamma \vdash_l C$.

Notice that executable nets are well-typed. Our main results will be stated in terms of executable nets; indeed, due to the dynamic acquisition of capabilities, well-formed nets that are statically deemed well-typed can still give rise to run-time errors. However, by marking those actions that should be checked at run-time, well-typed (and well-formed) nets can be transformed into executable nets that, instead, cannot give rise to run-time errors (see Corollary 3.3.11).

We now adapt the results in Section 3.2.3 to the new scenario. We only present the most relevant differences.

Lemma 3.3.6 (Weakening) *If $\Gamma \vdash_l^L C$ then $\Gamma \uplus \Gamma' \vdash_l^L C$.*

Lemma 3.3.7 (Substitutivity)

1. *If $\Gamma \vdash_l^L C$ then, for any substitution σ , $\Gamma\sigma \vdash_l^{L'}$ $C\sigma$ where $L' = L - \text{dom}(\sigma)$.*
2. *If $\Gamma \vdash_l^{L \cup \{X\}} C$, $X \notin L$ and $\Gamma \vdash_l^L Q$, then $\Gamma \vdash_l^L C[Q/X]$.*

Proof:

1. The proof is by induction on length of the inference of the type judgement. The base cases (i.e., rules (TD-NIL), (TD-DAT) and (TD-VAR)) are obvious. Let us examine the case in which the last rule used is (TD-RCV) (the cases for (TD-REC), (TD-PAR), (TD-SND), (TD-NEW) and (TD-MARK-SND/RCV) are similar or easier). By hypothesis, for some process Q and action a such that $\Gamma(\text{tgt}(a)) \sqsubseteq_{\Pi} \{\text{cap}(a)\} \subset \{i, r\}$, we have that $C = a.Q$ and $\Gamma \vdash_l^L a.Q$; thus, $\text{upd}(\Gamma, \text{arg}(a)) \vdash_l^{L \cup \text{bn}(\text{arg}(a))} Q$. Without loss of generality, we can assume that $\text{dom}(\sigma) \cap \text{bn}(\text{arg}(a)) = \emptyset$ (otherwise, if this is not the case, we could rename the bound names); thus we have $(a.Q)\sigma = a\sigma.Q\sigma$. Now, by induction, we have that $(\text{upd}(\Gamma, \text{arg}(a)))\sigma \vdash_l^{L''} Q\sigma$, where $L'' = (L \cup \text{bn}(\text{arg}(a))) - \text{dom}(\sigma) = (L - \text{dom}(\sigma)) \cup \text{bn}(\text{arg}(a)) = L' \cup \text{bn}(\text{arg}(a))$. Now, $\text{upd}(\Gamma\sigma, \text{arg}(a\sigma)) \vdash_l^{L''} Q\sigma$ (indeed, it is easy to prove that $(\Gamma_1 \uplus \Gamma_2)\sigma = (\Gamma_1\sigma) \uplus (\Gamma_2\sigma)$) and, by applying rule (TD-RCV), we conclude that $\Gamma\sigma \vdash_l^{L'} a\sigma.Q\sigma$, i.e. $\Gamma\sigma \vdash_l^{L'} (a.C)\sigma$.

2. The proof proceeds by induction on length of the inference of the type judgement. The base cases (i.e., rules (TD-NIL), (TD-DAT) and (TD-VAR)) are obvious. If the last used rule is (TD-REC), the thesis easily follows by induction (recall that substitution application may require renaming of bound variables to avoid capturing free names). If the last used rule is (TD-PAR), the thesis follows by induction using the fact that $(C_1|C_2)\sigma = C_1\sigma|C_2\sigma$. The cases when the last rule used is (TD-RCV), (TD-SND), (TD-NEW) or (TD-MARK-SND/RCV) are similar; we examine only the first one. By hypothesis, for some process P and action a such that $\Gamma(\text{tgt}(a)) \sqsubseteq_{\Pi} \{\text{cap}(a)\} \subset \{i, r\}$, we have that $C = a.P$ and $\Gamma \vdash_l^{L\cup\{X\}} a.P$; thus, for the premises of rule (TD-RCV), we have that $\text{upd}(\Gamma, \text{arg}(a)) \vdash_l^{L\cup\{X\}\cup\text{bn}(\text{arg}(a))} P$. By hypothesis, $\Gamma \vdash_l^L Q$ thus, by using Lemma 3.3.6, we have that $\text{upd}(\Gamma, \text{arg}(a)) \vdash_l^L Q$. Moreover, we can always assume that $\text{bn}(\text{arg}(a)) \cap \text{fn}(Q) = \emptyset$; hence, it also holds that $\text{upd}(\Gamma, \text{arg}(a)) \vdash_l^{L\cup\text{bn}(\text{arg}(a))} Q$. Then, by using the induction hypothesis, we get that $\text{upd}(\Gamma, \text{arg}(a)) \vdash_l^{L\cup\text{bn}(\text{arg}(a))} P[Q/X]$ and, by applying rule (TD-RCV), we obtain $\Gamma \vdash_l^L a.(P[Q/X])$, that is $\Gamma \vdash_l (a.P)[Q/X]$. ■

Now we prove that the property of a net of being executable is an invariant both of the structural congruence and of the reduction relation.

Lemma 3.3.8 *If N is executable and $N \equiv N'$ then N' is executable.*

Theorem 3.3.9 (Subject Reduction) *If N is executable and $N \mapsto N'$, then N' is executable.*

Proof: The proof proceeds by induction on the length of the inference of $N \mapsto N'$. We only present the most peculiar cases.

(RED-EVAL). Since by hypothesis N is executable, we have that $\delta \vdash_l \mathbf{eval}(Q)@l'.P$.

Due to the form of the process involved, rule (TD-SND) has been the last one applied to deduce the last type judgement; hence we also have that $\delta \vdash_l P$. Moreover, by applying (TD-PAR) to $\delta' \vdash_{l'} C'$ (that holds by hypothesis) and to $\delta' \vdash_{l'} Q \triangleright Q'$ (that is the premise of rule (RED-EVAL)), we get that $\delta' \vdash_{l'} C' | Q'$. This suffices to conclude that N' is executable.

(RED-IN). Since it holds that $\delta' \vdash_{l'} \mathbf{nil}$, we are only left to prove that $\delta \cup \delta'' \vdash_l P\sigma$.

Now, by hypothesis, we have that $\delta \vdash_l \mathbf{in}(T)@l'.P$, where rule (TD-RCV) has been the last one applied to infer the judgement; hence we also have that $\text{upd}(\delta, T) \vdash_l^{\text{bn}(T)} P$. By definition, if $\{x_i : \pi_i\}_{i \in I}$ are the formal fields of T , we have that $\text{upd}(\delta, T) = \delta \cup [x_i \mapsto \pi_i]_{i \in I}$. Moreover, by the premise of rule (RED-IN) and by Proposition 3.3.4, we have that $\text{match}_l^\delta(T, t) = \langle \delta'', \sigma \rangle$, where $\delta'' = [l_i \mapsto \pi_i]_{i \in I}$ and $\sigma = [l_i/x_i]_{i \in I}$. Now, $\text{upd}(\delta, T) = \delta \cup [x_i \mapsto \pi_i]_{i \in I}$ implies that $\text{upd}(\delta, T)\sigma = \delta \cup [l_i \mapsto \pi_i]_{i \in I} = \delta \cup \delta''$. Thus, by applying Lemma 3.3.7.1 to $\text{upd}(\delta, T) \vdash_l^{\text{bn}(T)} P$, we conclude that $\delta \cup \delta'' \vdash_l P\sigma$.

(RED-MARK). By hypothesis, we have that $\delta \vdash_l a.P$. Due to the form of the process involved in the judgement, rule (TD-MARK-SND) has been the last one applied to deduce the judgement; hence we also have that $upd(\delta, arg(a)) \vdash_l P$. By the premise of (RED-MARK), we have that, when the reduction takes place, $\delta(tgt(a)) \sqsubseteq_{\Pi} \{cap(a)\}$. Hence, by applying (TD-SND)/(TD-RCV), we can derive $\delta \vdash_l a.P$, that, together with the hypothesis $\delta' \vdash_{l'} C'$, implies that $l ::^{\delta} a.P \parallel l' ::^{\delta'} C'$ is executable. The thesis now follows by induction. ■

To prove type safety we still use the notion of run-time errors defined in Table 3.4. The rules are formally unchanged. Notice that, since marked actions are checked at run-time, they cannot give rise to run-time errors. At most, when their execution is not permissible, the process that is trying to execute them blocks waiting for the acquisition of the corresponding capabilities by a parallel process running at the same node.

Theorem 3.3.10 (Type Safety) *If N is executable then $N \nearrow$ does not hold.*

Proof: Similar to the proof of Theorem 3.2.10. ■

Therefore, we can combine together the results shown so far as in Section 3.2.3.

Corollary 3.3.11 (Global Type Soundness) *If N is executable and $N \mapsto^* N'$, then $N' \nearrow$ does not hold.*

Theorem 3.3.12 (Local Type Soundness) *Let N be a net. If N_S is executable and $N \mapsto^* N'$, then $N'_S \nearrow$ does not hold.*

3.3.3 Example: Subscribing On-line Publications

In this section, we take up the simple example presented at the end of Section 3.3.1 in order to show μ KLAIM's programming style and a way to exploit its type system for establishing access control policies.

Suppose that a user U wants to subscribe a 'licence' to enable accessing on-line publications of a given publisher P . To model this scenario we use three localities, l_U , l_P and l_C , respectively associated to U , P and to the repository containing P 's on-line accessible publications (this slightly differs from the informal scenario sketched in Section 3.3.1 – we explain later the reason why node l_C is needed). First of all, U sends a subscription request to P including its address (together with an 'out' capability) and credit card number; then, U waits for a tuple that will deliver it the capability r needed to access P 's publications and proceeds with the rest of its activity. The behaviour described so far is implemented by the process

$$A_U \triangleq \mathbf{out}(\text{"Subscr"}, l_U : [l_P \mapsto \{o\}], CrCrd) @ l_P. \mathbf{in}(\text{"Acc"}, !x : \{r\}) @ l_U. R$$

where process R may contain operations like $\mathbf{read}(\dots) @ l_C$.

Once P has received the subscription request and checked (by possibly using a third party authority) the validity of the payment information, it gives U a ‘read’ capability over l_C . P ’s behaviour is modelled by the following process.

$$A_P \triangleq \mathbf{rec} X. \left(\mathbf{in}(\text{“}Subscr\text{”}, !x : \{o\}, !y)@l_P. \right. \\ \left. \text{check credit card } y \text{ of } x \text{ and require the payment .} \right. \\ \left. \mathbf{out}(\text{“}Acc\text{”}, l_C : [x \mapsto \{r\}])@x \mid X \right)$$

Concretely, the capability r will be delivered to U for a limited period of time (for example, annual subscriptions would obtain read capabilities valid for one year) or for a limited number of accesses. In Section 3.3.4 we shall present some simple ways to implement these features in our setting.

For processes A_U and A_P to behave in the expected way, the underlying net architecture, namely distribution of processes and access control policies, must be appropriately configured. A suitable net is:

$$l_U :: [l_U \mapsto \{o, i, r, e\}, l_P \mapsto \{o\}] \underline{A_U} \parallel l_P :: [l_P \mapsto \{o, i, r, e\}, l_C \mapsto \{o, i, r\}] A_P \\ \parallel l_C :: [^1] \langle paper1 \rangle \mid \langle paper2 \rangle \mid \dots$$

where we have intentionally used $\underline{A_U}$ to emphasise the fact that the static type checking might have marked some actions occurring in A_U , e.g. actions $\mathbf{read}(\dots)@l_C$ in R . Upon completion of the protocol, the net will be

$$l_U :: [l_U \mapsto \{o, i, r, e\}, l_P \mapsto \{o\}, l_C \mapsto \{r\}] \underline{R} \parallel l_P :: [l_P \mapsto \{o, i, r, e\}, l_C \mapsto \{o, i, r\}, l_U \mapsto \{o\}] A_P \\ \parallel l_C :: [^1] \langle paper1 \rangle \mid \langle paper2 \rangle \mid \dots$$

To conclude this section, we want to remark four features of this example that shed light on some peculiarities of our framework.

1. P ’s papers cannot be safely put in l_P ’s TS because otherwise the integrity of P ’s publications could be compromised by the execution in l_U of the legal process $\mathbf{out}(\text{not_a_P’s_paper})@l_P$. Indeed, differently from Section 3.4 (see later on), our types do not provide support for restricting the kind of tuples over which actions can operate thus enabling $\mathbf{out}(\text{“}Subscr\text{”}, l_U : [l_P \mapsto \{o\}], CrCrd)@l_P$ and disabling $\mathbf{out}(\text{not_a_P’s_paper})@l_P$.
2. Knowledge of address l_C is not enough for reading papers: the ‘read’ capability is needed. Indeed, security in the μ KLAIM framework does not rely on name knowledge but on access control policies.
3. Once the ‘read’ capability over l_C has been acquired, all processes eventually spawned at l_U can access P ’s on-line publications. In other terms, U obtains a sort of ‘site licence’ valid for all processes running at l_U . This fact should not be considered as a security breach: indeed, in order to enter in l_U , a mobile process could be required to exhibit some credential (like a password [107]), that however we do not model in our framework. Moreover, notice that this way of handling privileges is different from [59], where, by using

the same protocol, U would have obtained a sort of ‘individual licence’ for process R . In the next section we will present variations of our framework that permit delivering different capabilities to co-located processes.

4. The licence delivered by P to U can be used only at l_U since the granting associated to l_C only delivers to l_U the capability r over l_C . Moreover, no intruder can interfere with the protocol between the user and the publisher because the tuple $\langle \text{“Acc”}, l_C : [l_U \mapsto \{r\}] \rangle$ located at l_U can only be retrieved by processes running at l_U (see rules (M₁) and (M₂) in Table 3.7). A similar argument holds for the tuple $\langle \text{“Subscr”}, \dots \rangle$ inserted by A_U at l_P .

3.3.4 Variations on Capability Management

Up to now, capabilities are always acquired by the node hosting the process performing actions **in/read**, and not by the process itself. This may be adequate in some cases, e.g. when a department subscribes a ‘site licence’ (i.e. valid for all its members), and unrealistic in others, e.g. when a mobile process has to buy a good on behalf of its owner. Moreover, capabilities can only increase; this does not fit well to control wastable resources where one usually wants to count the number of times a given resource is used or to deliver accesses for a limited period of time.

In the next two subsections, we will show that our framework can be smoothly tailored for taking into account these different scenarios. For each variant, we shall first describe the scenario we want to model from an operational point of view (that should be more intuitive and that enables the presentation of a concrete example motivating the variation). Then, we shall discuss how the typing theory can be accommodated to keep the results of Section 3.3.2 valid.

Variations on Capabilities Acquisition

In this section, we show an adaption of our framework that allows processes to acquire capabilities for themselves. We start by presenting a scenario where all the dynamically acquired capabilities are assigned to processes; then, we shall combine together the possibility of assigning capabilities to processes and to nodes.

(I) Acquisition by Processes. We start modifying our framework in such a way that capabilities, in particular those dynamically acquired, can be associated to processes. To this aim, we annotate located processes with a type that specifies the capabilities they own. Thus, a process can also use its own private capabilities, in addition to the capabilities of the executing node that are shared by all co-located processes. Now, a μ KLAIM node is of the form $l ::^\delta \mathcal{AC}$, where \mathcal{AC} is an *annotated component* generated from the following auxiliary syntactic productions

$$\mathcal{AC} ::= \langle t \rangle \mid \{\{ P \}\}_\delta \mid \mathcal{AC}_1 | \mathcal{AC}_2$$

Notice that only process components can be annotated.

<p>(RED-OUT')</p> $\frac{\llbracket t \rrbracket_{\delta \cup \delta_1}}{l ::^\delta \{ \mathbf{out}(t)@l'.P \}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto l ::^\delta \{ P \}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \langle t \rangle}$
<p>(RED-EVAL')</p> $\frac{\delta'[\delta_1] \vdash_{l'} Q \triangleright \underline{Q}}{l ::^\delta \{ \mathbf{eval}(Q)@l'.P \}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto l ::^\delta \{ P \}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \{ \underline{Q} \}_{\delta_1}}$
<p>(RED-IN')</p> $\frac{\mathit{match}_l^{\delta \cup \delta_1}(T, t) = \langle \delta'', \sigma \rangle}{l ::^\delta \{ \mathbf{in}(T)@l'.P \}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^\delta \{ P\sigma \}_{\delta_1 \cup \delta''} \parallel l' ::^{\delta'} \mathbf{nil}}$
<p>(RED-READ')</p> $\frac{\mathit{match}_l^{\delta \cup \delta_1}(T, t) = \langle \delta'', \sigma \rangle}{l ::^\delta \{ \mathbf{read}(T)@l'.P \}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^\delta \{ P\sigma \}_{\delta_1 \cup \delta''} \parallel l' ::^{\delta'} \langle t \rangle}$
<p>(RED-NEW')</p> $\frac{}{l ::^\delta \{ \mathbf{new}(l' : \delta').P \}_{\delta_1} \mapsto (\nu l')(l ::^\delta \{ P \}_{\delta_1 \cup [l' \mapsto \delta_1(l)]} \parallel l' ::^{\delta'} \mathbf{nil})}$
<p>(RED-MARK')</p> $\frac{l' = \mathit{tgt}(a) \quad \delta_1(l') \sqsubseteq_{\Pi} \{ \mathit{cap}(a) \} \quad l ::^\delta \{ a.P \}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto N}{l ::^\delta \{ \underline{a}.P \}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto N}$
<p>(RED-SPLIT'₁)</p> $\frac{l ::^\delta \mathcal{A}C_1 \parallel l ::^\delta \mathcal{A}C_2 \parallel N \mapsto l ::^\delta \mathcal{A}C'_1 \parallel l ::^\delta \mathcal{A}C'_2 \parallel N'}{l ::^\delta \mathcal{A}C_1 \mathcal{A}C_2 \parallel N \mapsto l ::^\delta \mathcal{A}C'_1 \mathcal{A}C'_2 \parallel N'}$
<p>(RED-SPLIT'₂)</p> $\frac{l ::^\delta \{ P \}_{\delta_1} \parallel l ::^\delta \{ Q \}_{\delta_1} \parallel N \mapsto l ::^\delta \{ P' \}_{\delta_2} \parallel l ::^\delta \{ Q \}_{\delta_1} \parallel N'}{l ::^\delta \{ P Q \}_{\delta_1} \parallel N \mapsto l ::^\delta \{ P' \}_{\delta_2} \{ Q \}_{\delta_1} \parallel N}$
<p>plus rules (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 3.8. $\llbracket \cdot \rrbracket_{_}$ is defined in Table 3.6 and $\mathit{match}_l^{\delta}(\cdot, \cdot)$ is defined in Table 3.7</p>

Table 3.9: Acquisition by Processes: Operational Semantics

The structural congruence is modified by replacing rules (STR-ALPHA) and (STR-ABS) in Table 2.2 with rules

$$\frac{(\text{ALPHA}') \quad P =_{\alpha} P'}{l ::^\delta \{ P \}_{\delta'} \equiv l ::^\delta \{ P' \}_{\delta'}} \quad \frac{(\text{STR-ABS}')}{l ::^\delta \mathcal{A}C \equiv l ::^\delta \mathcal{A}C | \{ \mathbf{nil} \}_{\delta'}}$$

Instead, the operational semantics is changed for managing the acquisition of capabilities that now increases process type annotations while leaves types of nodes unchanged. In the initial configuration, all processes could have assigned the same empty type or could have assigned different and more informative type annotations, reflecting different capabilities for the processes. Table 3.9 presents the modified reduction rules that should be self-explicative. Notice that in (RED-MARK') only the type associated to the process is used to check admissibility of action a ; indeed, the type of the node does never change (i.e. it is statically known) and has already been used in the static typing phase.

Let us now briefly revise the subscription example. If in the initial configuration all processes have assigned the empty type, the evolution of the net according to the modified semantics leads to

$$l_U :: [l_U \mapsto \{o, i, r, e\}, l_P \mapsto \{o\}] \{ \underline{R} \}_{[l_C \mapsto \{r\}]} \parallel l_P :: [l_P \mapsto \{o, i, r, e\}, l_C \mapsto \{o, i, r\}] A_P \\ \parallel l_C :: [1] \langle paper1 \rangle \mid \langle paper2 \rangle \mid \dots$$

where now \underline{R} is the only process having the capability to access the papers stored at l_C . Moreover, notice that the capability o over l_U delivered by A_U to A_P disappears upon completion of the parallel component running at l_P that handles A_U 's request. Indeed, at the end of its task such a component becomes $\{ \mathbf{nil} \}_{[l_U \mapsto \{o\}]}$ and hence can be removed by using rule (STR-ABS') above.

(II) Acquisition by Nodes and Processes. In practice, a (mobile) process could acquire some capabilities and, from time to time, decide whether it wants to keep them for itself or to share them with other processes running at the same node. An example of the first scenario is a mobile process that buys a book and obviously wants the book to be delivered to its owner. An example of the second scenario is a process that subscribes a 'site licence' to enable accessing some on-line publications to be shared with all members of a given department.

A simple way to model both cases is to use different acquisition actions depending on whether the acquisition should be made on behalf of the node or of the process. Hence, we could leave the operational semantics of actions **in/read** unchanged (i.e. as given in Section 3.3.1) apart for the replacement of processes with annotated processes, add actions **inpr**(T)@ u and **readpr**(T)@ u to the syntax, and model their operational semantics by using rules akin to (RED-IN') and (RED-READ') in Table 3.9. In such a way, actions **in/read** would increase the type of the node where they are executed while actions **inpr/readpr** would increase the private type of the executing process. Therefore, in this variant of the calculus, processes are more powerful because, whenever capabilities are acquired, they can decide to share these capabilities with the other co-located processes or not.

Of course, to control the new actions, we also need to introduce the corresponding capabilities and to extend the capability ordering relation. Finally, notice that, since node types can dynamically change (like in the original semantics), in rule (RED-MARK') the hypothesis $\delta_1(l') \sqsubseteq_{\Pi} \{cap(a)\}$ must be replaced by

$\delta_1(l') \cup \delta(l') \sqsubseteq_{\Pi} \{cap(a)\}$. Indeed, a marked action can be enabled both by the capabilities accumulated by the process and by the capabilities offered by the hosting node.

Type Soundness. We now sketch how the results of Section 3.3.2 can be adapted to the variant we have just presented (notice that the setting of Section 3.3.4(I) is clearly an instance of the theory we develop here). The static typing needs smooth extensions: it should consider annotated processes and it should let rule (TD-RCV) deal with actions **inpr/readpr** too. The first task can be carried on by adding the following rule

$$(TD-ANN) \quad \frac{\Gamma \uplus \delta \vdash_l^L P \triangleright \underline{P}}{\Gamma \vdash_l^L \{\{P\}\}_\delta \triangleright \{\{\underline{P}\}\}_\delta}$$

A marked annotated component $\underline{\mathcal{AC}}$ is an annotated component that may contain annotated marked processes of the form $\{\{P\}\}_\delta$. Then, well-typedness and executableness are defined like before, but take into account annotated components.

Definition 3.3.13 *A net is well-typed if, for each node $l ::^\delta \mathcal{AC}$, there exists a component $\underline{\mathcal{AC}}$ such that $\delta \vdash_l \mathcal{AC} \triangleright \underline{\mathcal{AC}}$. A net is executable if, for each node $l ::^\delta \mathcal{AC}$, it holds that $\delta \vdash_l \mathcal{AC} \triangleright \mathcal{AC}$ (abbreviated as $\delta \vdash_l \mathcal{AC}$).*

Finally, run-time errors are defined accordingly, by letting rule (ERRACT) become

$$(ERRACT') \quad \frac{\delta(tgt(a)) \cup \delta'(tgt(a)) \not\sqsubseteq_{\Pi} \{cap(a)\}}{l ::^\delta \{a.P\}_{\delta'} \nearrow}$$

Thus, soundness of the revised framework can be formulated and proved like in Theorem 3.3.12. Notice that, by taking the set S to be $fn(N)$, we easily obtain also the global type soundness of Corollary 3.3.11.

Managing Loss of Capabilities

In this subsection, we deal with some scenarios where capabilities can be lost. The three settings we shall present mainly differ in the formal definition of capabilities and in the way in which capabilities are lost. The main common feature is that static typing is weakened since there are a lot of ingredients that can dynamically change. As it could be expected, more flexibility requires more run-time checks.

Since in this subsection we need to express capabilities removal, we introduce notation $\delta = \delta_1, \delta_2$ to state that, for each $u \in \mathcal{L}$, we have $\delta(u) = \delta_1(u) \cup \delta_2(u)$, $\delta_1(u) = \delta(u) - \delta_2(u)$ and $\delta_2(u) = \delta(u) - \delta_1(u)$. A similar notation is exploited also for grantings.

<p>(RED-OUT'')</p> $\frac{\llbracket t \rrbracket_{\delta_1}^{\delta_1} = \delta_1'}{l ::^\delta \{\{\mathbf{out}(t)@l'.P\}\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto l ::^\delta \{\{P\}\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \langle t \rangle}$ <p>(RED-EVAL'')</p> $\frac{\delta_1 = \delta_1', \delta_1'' \quad \delta' \vdash_{l'} Q \triangleright \underline{Q}}{l ::^\delta \{\{\mathbf{eval}(Q)@l'.P\}\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto l ::^\delta \{\{P\}\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \{\{\underline{Q}\}\}_{\delta_1''}}$ <p>(RED-IN'')</p> $\frac{\mathit{match}_l^{\delta \cup \delta_1}(T, t) = \langle \delta'', \sigma, t' \rangle}{l ::^\delta \{\{\mathbf{in}(T)@l'.P\}\}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^\delta \{\{P\sigma\}\}_{\delta_1 \cup \delta''} \parallel l' ::^{\delta'} \mathbf{nil}}$ <p>(RED-READ'')</p> $\frac{\mathit{match}_l^{\delta \cup \delta_1}(T, t) = \langle \delta'', \sigma, t' \rangle}{l ::^\delta \{\{\mathbf{read}(T)@l'.P\}\}_{\delta_1} \parallel l' ::^{\delta'} \langle t \rangle \mapsto l ::^\delta \{\{P\sigma\}\}_{\delta_1 \cup \delta''} \parallel l' ::^{\delta'} \langle t' \rangle}$ <p>(RED-MARK'')</p> $\frac{l' = \mathit{tgt}(a) \quad \delta_1 = \delta_1', [l' \mapsto \{cap(a)\}] \quad l ::^\delta \{\{a.P\}\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto N}{l ::^\delta \{\{\underline{a}.P\}\}_{\delta_1} \parallel l' ::^{\delta'} \mathcal{A}C' \mapsto N}$ <p>plus rules (RED-NEW'), (RED-SPLIT'₁) and (RED-SPLIT'₂) from Table 3.9 and rules (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 2.3</p>

Table 3.10: Consumption of Capabilities: Operational Semantics

Consumption. If we interpret the ‘acquisition of capabilities’ as the ‘purchase of services/goods’, it seems natural that a process will lose the acquired capability once it uses the service. For example, by paying the price of a book a user purchases one copy of the book; if he wants another copy, he has to pay again. To enable multiple acquisitions and consumptions of capabilities, we should be able to count the number of capabilities that nodes/processes have over each resource (this is somehow similar to ‘affine’ capability types of [38]). To this aim, we modify our semantic framework by working with *multisets* of capabilities, instead of sets; in particular, Π now denotes the set of the multisets built upon $\{i, e, r, o\}$. All the operations over and relations between sets used up-to now (i.e., union, subset inclusion, ...) must be considered as operations over and relations between multisets.

We start considering the case of dynamic acquisition and consumption of capabilities only by processes (see Section 3.3.4(I)). This means that node types are statically known and left unchanged by the operational semantics. The operational rules are modified as reported in Table 3.10. The main change is that process

capabilities must be deleted whenever used; this happens in rules (RED-OUT''), (RED-EVAL'') and (RED-MARK''). Notice that the checking of grantings now deletes the process capabilities passed in the tuple and returns the capabilities left. Its formal definition updates Table 3.6 as follows:

$$\frac{\begin{array}{l} \mu = [l_i \mapsto \pi_i]_{i=1,\dots,k} \\ \delta_1 = \delta'_1, [l \mapsto \bigcup_{i=1}^k (\pi_i - \delta(l))] \\ \forall i = 1, \dots, k. (\delta(l) \cup \delta_1(l)) \sqsubseteq_{\Pi} \pi_i \end{array}}{\llbracket l : \mu \rrbracket_{\delta}^{\delta_1} = \delta'_1} \quad \frac{\begin{array}{l} \llbracket t_1 \rrbracket_{\delta}^{\delta_1} = \delta_2 \\ \llbracket t_2 \rrbracket_{\delta}^{\delta_2} = \delta_3 \end{array}}{\llbracket t_1, t_2 \rrbracket_{\delta}^{\delta_1} = \delta_3}$$

Finally, also pattern matching needs to be modified; now, when it is invoked by l on T and t , it returns a triple $\langle \delta'', \sigma, t' \rangle$. The difference is in the tuple t' obtained by removing from the grantings within t all the capabilities granted to l (i.e., the capabilities collected in δ''). This is necessary otherwise repeated accesses to a tuple via actions **read** would lead to a form of ‘capability forging’. Indeed, each time a process at l reads t , the capabilities in δ'' would be delivered to the process. Since the **read** can be repeated an unbounded number of times (until $\langle t \rangle$ is available), it would be possible to acquire several times the capabilities in δ'' . The new formulation of function *match* relies on the following modification of rule (M₂)

$$\frac{\delta(l') \cup \mu(l) \sqsubseteq_{\Pi} \pi \quad \mu = \mu', [l \mapsto (\pi - \delta(l'))]}{\mathit{match}_l^{\delta}(!x : \pi, l' : \mu) = \langle [l' \mapsto (\pi - \delta(l'))], [l'/x], l' : \mu' \rangle}$$

where only the capabilities delivered by the tuple that are not already owned by the executing node are used to enrich the policy of the executing process (this is needed to avoid delivering the process capabilities already in δ). As concerns (M₁), it is modified to additionally return the tuple passed as second argument to function $\mathit{match}_l^{\delta}$, while (M₃) is modified to additionally return the tuple resulting from the concatenation of the two tuples returned by its premises.

Taking up the example of Section 3.3.3, we can now program the acquisition (and the consumption) of a fixed number of capabilities r over the on-line repository. The user explicitly requires a number k of capabilities r and the publisher will charge on U 's credit card the cost of k accesses to its publications. The processes implementing these behaviours are

$$\begin{aligned} A_U &\triangleq \mathbf{out}(\text{“Subscr”}, l_U : [l_P \mapsto \{o\}], CrCrd, k) @ l_P. \\ &\quad \mathbf{in}(\text{“Acc”}, !x : \{k \times r\}) @ l_U.R \\ A_P &\triangleq \mathbf{rec} X. (\mathbf{in}(\text{“Subscr”}, !x : \{o\}, !y, !z) @ l_P. \\ &\quad \text{check credit card } y \text{ of } x \text{ and charge the cost for } z \text{ accesses} . \\ &\quad \mathbf{out}(\text{“Acc”}, l_C : [x \mapsto \{z \times r\}]) @ x \mid X) \end{aligned}$$

where $\{_ \times r\}$ stands for the multiset with $_$ occurrences of capability r .

Type Soundness. Differently from Section 3.3.4, process capabilities do not play any role in the static typing (thus, rule (TD-ANN) is missing): indeed, since they

can also decrease, it is statically impossible to rely on them to determine whether a given action will be legal at run-time or not. As an example, consider the net $l ::= [1] \{ P|Q \}_{[l' \mapsto \{o\}]}$, where $P \triangleq \mathbf{out}(t)@l'$ and $Q \triangleq \mathbf{out}(t')@l'$. In this case, exactly one between P and Q will be able to perform action \mathbf{out} while the other process will be blocked, depending on the execution order. However, it is impossible to statically tell which one will evolve and which one will get stuck (and hence both of them have to be marked). Moreover, the static semantics differs from that presented before because it has to mark all the actions, except those directly enabled by the access policy of the node where the inference takes place (i.e. those actions a such that $\delta(\mathit{tgt}(a)) \sqsubseteq_{\Pi} \{cap(a)\}$, where δ is the type of the node). This is necessary to properly handle nodes like $l ::= [l' \mapsto \{i\}] \mathbf{in}(!u : \{o\})@l'.\mathbf{out}(\cdot)@l'.\mathbf{out}(\cdot)@u$, where action \mathbf{in} should be the only unmarked one after static typing. Indeed, if we use the static typing of Section 3.3.1, the second action \mathbf{out} would not be marked; this could generate a run-time error because if u is replaced by l' upon execution of the \mathbf{in} , then the acquired capability o (that enables the execution of the second action \mathbf{out}) would be consumed to perform the first action \mathbf{out} .

Well-typed and executable nets are formally defined like in Definition 3.3.13; recall that process types are never used to infer judgements, because rule (TD-ANN) is missing here. Run-time errors are defined like in Section 3.3.4(II), i.e. by exploiting rule (ERRACT'). Thus, type soundness can be still stated and proved similarly to Theorem 3.3.12.

A more general framework. Finally, let us now briefly consider the general setting where both processes and nodes can dynamically acquire and consume capabilities (see Section 3.3.4(II)). This scenario is the most expensive because a static typing phase cannot be exploited at all and all actions must be checked at run-time. In fact, since also node types can dynamically change, it is impossible to statically determine if a given action will have the necessary capabilities at run-time. Moreover, both the type associated to a process and the type of the node where the process is running can provide the process with the capability necessary to perform a given action. In this case, the capability can be removed from the type of the node or from the type of the process, and a strategy must be implemented. The operational rules can be easily modified to control capabilities and remove the used ones, but, to save space, we do not show the modified rules.

Validity duration. Another possible way of modelling capability lost is by introducing duration, as we already mentioned in the example of Section 3.3.3. Each capability can be assigned a validity duration by indexing it with a natural number (or ∞) representing a period of time during which the capability is valid, i.e. can be used: a capability is available until its validity has not been expired. Thus, types (and grantings) map \mathcal{L} to Π' , where Π' is the powerset of $C \times (\mathbf{Nat} \cup \{\infty\})$ and it is ranged over by p . For example, $[l \mapsto \{i_{10}, o_5, e_{\infty}\}]$ expresses the fact that it is still possible to perform over l actions \mathbf{in} for 10 time units, actions \mathbf{out} for 5 time units and actions \mathbf{eval} for ever. A capability associated to ∞ is called ‘persistent’

since its validity never expires; notationally, we write them without any annotation (notice that all the capabilities considered so far were indeed persistent).

The operational semantics of the basic framework needs to be modified to model time passing and the effect of time passing on validity durations. To point out the passing of τ time units, we label net reductions with τ . All the rules in Table 3.8 except (RED-PAR) and (RED-STRUCT) represent computational steps and are assumed to be instantaneous; thus, the reductions occurring therein are labelled with '0'. The reductions contained in rules (RED-PAR) and (RED-STRUCT) are instead labelled with a generic label τ . Of course, this choice is far from being realistic but it allows us to incorporate durations in our framework with very small modifications and can model all real situations. Because of the intrinsic asynchronous nature of our nets, we assume that time can pass differently in different parts of the net but, at each node, time passes uniformly for all the processes running there. This behaviour is implemented by adding to the rules in Table 3.8 the following one

$$\text{(RED-TIME)} \quad l ::^\delta C \xrightarrow{\tau} l ::^{(\delta)-\tau} (C)_{-\tau}$$

Function $(\cdot)_{-\tau}$ is defined inductively as

$$(C_1|C_2)_{-\tau} = (C_1)_{-\tau} | (C_2)_{-\tau}$$

$$\langle\langle t \rangle\rangle_{-\tau} = \langle t' \rangle \quad \text{with } t' \text{ obtained from } t \text{ by replacing each } \mu \text{ with } (\mu)_{-\tau}$$

$$[]_{-\tau} = []$$

$$([l \mapsto p])_{-\tau} = [l \mapsto p']$$

where p' is obtained from p by:

- subtracting τ to all the durations, and
- deleting the capabilities with a non-positive duration

$$(\delta \uplus \delta')_{-\tau} = (\delta)_{-\tau} \uplus (\delta')_{-\tau}$$

$$(\mu \uplus \mu')_{-\tau} = (\mu)_{-\tau} \uplus (\mu')_{-\tau}$$

and it is the identity function in all the other cases. Thus, it can be easily seen that when τ_1 time units pass in l_1 and τ_2 time units pass in l_2 , the net $l_1 ::^{\delta_1} C_1 \parallel l_2 ::^{\delta_2} C_2$ evolves as follows:

$$\begin{aligned} l_1 ::^{\delta_1} C_1 \parallel l_2 ::^{\delta_2} C_2 &\xrightarrow{\tau_1} l_1 ::^{(\delta_1)-\tau_1} (C_1)_{-\tau_1} \parallel l_2 ::^{\delta_2} C_2 \\ &\xrightarrow{\tau_2} l_1 ::^{(\delta_1)-\tau_1} (C_1)_{-\tau_1} \parallel l_2 ::^{(\delta_2)-\tau_2} (C_2)_{-\tau_2} \end{aligned}$$

Type Soundness. We can statically control only the operations that are enabled by persistent capabilities; all the other operations have to be marked, since it is not possible to exactly know when they will be performed. In particular, all the actions having a variable as target must be marked. Moreover, to avoid forging capability durations, we also need to ensure that a process delivers a capability with duration τ only if the capability is persistent or has a duration at least τ in the type of the node where the process runs.

These tasks can be achieved by defining an ordering on Π' , written $\sqsubseteq_{\Pi'}$, as follows

$$\frac{\tau \geq \tau'}{\{c_\tau\} \sqsubseteq_{\Pi'} \{c_{\tau'}\}} \quad \frac{p_1 \subseteq p_2}{p_2 \sqsubseteq_{\Pi'} p_1} \quad \frac{p_1 \sqsubseteq_{\Pi'} p'_1 \quad p_2 \sqsubseteq_{\Pi'} p'_2}{(p_1 \cup p_2) \sqsubseteq_{\Pi'} (p'_1 \cup p'_2)}$$

Clearly $\llbracket \cdot \rrbracket_\delta$, $match_l^\delta(\cdot, \cdot)$ and $mark_L(\Gamma)$ now exploit this ordering. In particular, this fact implies that, since $cap(a)$ returns a capability that is *not* annotated, action a is marked whenever a corresponding persistent capability is missing in the current typing environment. On the other hand, rule (RED-MARK) still invokes \sqsubseteq_{Π} , that can be straightforwardly extended to annotated capabilities by ignoring durations.

Well-typedness and executableness are still defined like in Definitions 3.3.3 and 3.3.5. Type soundness is then formulated and proved like in Corollary 3.3.11 and Theorem 3.3.12: it relies on the run-time errors defined in Table 3.4, that are still defined in terms of \sqsubset_{Π} (properly extended to ignore validity durations). The only difference is that, in stating and proving subject reduction (Theorem 3.3.9), we also need to consider time passing, i.e. reductions of the form $\xrightarrow{\tau}$. We omit the easy details.

Revocation. We shall now touch upon a new scenario where capabilities can be revoked, i.e. a node can delete capabilities of other nodes. To rule out obvious nasty attacks, we allow l to remove a type δ from l' only if l has previously passed a supertype of δ to l' (notice that this complies with standard trends in *discretionary access control models*). In doing so, we have also to take into account the fact that several nodes could have passed δ to l' .

We let \mathcal{S} to be the set of the finite subsets of \mathcal{L} and we let s, s', \dots to range over \mathcal{S} . We now annotate capabilities, ranged over by c , with the identity of the deliverers, thus obtaining the set of *annotated capabilities* Π' , ranged over by p . Formally, Π' contains the subsets of $\{r, i, o, e\} \times \mathcal{S}$ such that, if $(c_1, s_1) \in p$ and $(c_2, s_2) \in p$, for some $s_1 \neq s_2$, then $c_1 \neq c_2$. Statically assigned capabilities take the form (c, \emptyset) and are abbreviated as c . We let the preorder $\sqsubseteq_{\Pi'}$ on annotated capabilities to be defined by the following rules:

$$\frac{s_2 \subseteq s_1 \vee s_1 = \emptyset}{\{(c, s_1)\} \sqsubseteq_{\Pi'} \{(c, s_2)\}} \quad \frac{p_1 \subseteq p_2}{p_2 \sqsubseteq_{\Pi'} p_1} \quad \frac{p_1 \sqsubseteq_{\Pi'} p'_1 \quad p_2 \sqsubseteq_{\Pi'} p'_2}{(p_1 \cup p_2) \sqsubseteq_{\Pi'} (p'_1 \cup p'_2)}$$

Grantings are left unchanged, i.e. they are finite partial functions from \mathcal{L} to Π , while types now use annotated capabilities. We use γ to range over these annotated types that, formally, are functions mapping \mathcal{L} to Π' such that $\gamma(l) \neq \emptyset$ only for finitely many l s. E.g., the type $[l \mapsto \{(i, \{l_1\}), (o, \{l_2, l_3\})\}]$ used as access control policy of node l' enables actions **in/out** from l' over l , and records that the capability i has been delivered by l_1 while the capability o has been delivered by both l_2 and l_3 . The subtyping relation between annotated types, \leq' , is defined like \leq but of course relies on $\sqsubseteq_{\Pi'}$ instead of \sqsubseteq_{Π} . If γ_1 and γ_2 are annotated types, the extension

(RED-OUT''')	$\frac{\llbracket t \rrbracket_{pol(\gamma)}}{l ::^\gamma \mathbf{out}(t)@l'.P \parallel l' ::^{\gamma'} C' \mapsto l ::^\gamma P \parallel l' ::^{\gamma'} C' \langle t \rangle^l}$
(RED-EVAL''')	$\frac{pol(static(\gamma')) \vdash_P Q \triangleright \underline{Q}}{l ::^\gamma \mathbf{eval}(Q)@l'.P \parallel l' ::^{\gamma'} C' \mapsto l ::^\gamma P \parallel l' ::^{\gamma'} C' \underline{Q}}$
(RED-IN''')	$\frac{match_l^{pol(\gamma)}(T, t) = \langle \delta, \sigma \rangle}{l ::^\delta \mathbf{in}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle^{l''} \mapsto l ::^{\gamma \cup \delta^{l''}} P\sigma \parallel l' ::^{\delta'} \mathbf{nil}}$
(RED-READ''')	$\frac{match_l^{pol(\gamma)}(T, t) = \langle \delta, \sigma \rangle}{l ::^\delta \mathbf{read}(T)@l'.P \parallel l' ::^{\delta'} \langle t \rangle^{l''} \mapsto l ::^{\gamma \cup \delta^{l''}} P\sigma \parallel l' ::^{\delta'} \langle t \rangle^{l''}}$
(RED-REVOKE)	$\frac{\gamma' = \gamma'', \delta^{l\}}{l ::^\gamma \mathbf{revoke}(\delta)@l'.P \parallel l' ::^{\gamma'} C' \mapsto l ::^\gamma P \parallel l' ::^{\gamma''} C'}$
<p>plus rules (RED-NEW), (RED-SPLIT), (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 3.8, with γ in place of δ everywhere.</p> <p>$\llbracket \cdot \rrbracket_-$ is defined in Table 3.6 and $match_l^{\cdot}(\cdot, \cdot)$ is defined in Table 3.7</p>	

Table 3.11: Revocation of Capabilities: Operational Semantics

$\gamma_1 \cup \gamma_2$ is the annotated type γ' such that $\gamma'(u) = \gamma_1(u) + \gamma_2(u)$ for each $u \in \mathcal{L}$, where $p_1 + p_2$ is inductively defined as

$$\begin{aligned} \emptyset + p &= p \\ \{(c, s)\} + p &= \begin{cases} \{(c, s \uplus s')\} \cup p' & \text{if } (c, s') \in p \text{ and } p' = p - \{(c, s')\} \\ \{(c, s)\} \cup p & \text{if } (c, -) \notin p \end{cases} \\ (\{(c, s)\} \cup p) + p' &= \{(c, s)\} + (p + p') \end{aligned}$$

We let $s_1 \uplus s_2$ be $s_1 \cup s_2$ if both $s_i \neq \emptyset$, and \emptyset otherwise. Underlying the definition of \uplus there is the assumption that, if a capability has been statically assigned to a given node (and hence one of the s_i is the empty set), then no other node will ever be allowed to revoke it; a similar motivation inspired us the definition of \sqsubseteq_{IV} .

To enable capability revocations, we add operation $\mathbf{revoke}(\delta)@l$ to the syntax of μKLAIM actions. The new operational semantics is given in Table 3.11. The main modification w.r.t. Section 3.3.1 is a way to ‘sign’ a tuple with the identity of the producer, so that, when capabilities contained in the tuple are acquired, the identity of the granter is properly recorded to enable their possible future revocation. This can be obtained by letting located tuples take the form

$$\langle t \rangle^l$$

where l is the producer of the tuple. Then, when a policy is updated by exploiting

capabilities passed by a node l'' (see rules (RED-IN'') and (RED-READ'')), the received capabilities are annotated with l'' , as expressed by notation $\delta^{l''}$. Formally, for any $u \in \mathcal{L}$, we let $\delta^{l''}(u) = \{(c, \{l''\}) : c \in \delta(u)\}$. In the first four rules of Table 3.11, we used function $pol(\gamma)$ that yields a simple (i.e., not annotated) type δ by deleting from γ all capability annotations, and $static(\gamma)$, that is the annotated type obtained from γ by removing all the capabilities that have not been statically assigned. Finally, rule (RED-REVOKE) deals with revocations: it verifies that the revoked capabilities, δ , are present in the type γ' of l' , and that l was one of the grantors of δ in γ' .

We now show two possible uses of **revoke** in the example of Section 3.3.3. The first use consists in an alternative way of implementing the subscription for a fixed period of time d . Indeed, if we do not introduce validity durations as previously shown, we can let P to manage timing information: once U 's capability r has expired, P can revoke it. A simplified process A_P implementing this behaviour is

$$\begin{aligned}
A_P &\triangleq \mathbf{rec} X. (X \mid \mathbf{in}(\text{“Subscr”}, !x : \{o\}, !y, !d)@l_P. \\
&\quad \text{check } c.c. \text{ } y \text{ of } x \text{ and require the payment for duration } d. \\
&\quad \mathbf{out}(\text{“Acc”}, l_C : [x \mapsto \{r\}])@x. \mathbf{out}(x, \text{Today}() + d)@l'_P.B) \\
B &\triangleq \mathbf{rec} Y. \mathbf{in}(x, !s)@l'_P. \\
&\quad \mathbf{out}(\text{“check”}, x, \text{Today}(), \text{Today}() \leq s)@l'_P. \\
&\quad (\mathbf{in}(\text{“check”}, x, \text{Today}(), \mathbf{false})@l'_P. \mathbf{revoke}([l_C \mapsto \{r\}])@x \\
&\quad \mid \mathbf{in}(\text{“check”}, x, \text{Today}(), \mathbf{true})@l'_P. \mathbf{out}(x, s)@l'_P.Y)
\end{aligned}$$

where l'_P is a reserved locality where P stores timing information (we have silently used basic values representing dates and booleans, together with some obvious operations over them). Intuitively, process A_P handles timing expirations by recording in l'_P the expiration date of U 's subscription (given by function $\text{Today}() + d$). Then, process B repeatedly verifies the validity of the subscription by checking whether the current date (given by function $\text{Today}()$) is antecedent to the expiration date of U 's subscription. When expired, the capability enabling the access to P 's papers is revoked.

Another possible use of **revoke** in our example consists in revoking the access capability to a misbehaved user, e.g. a user that sold the acquired capability r to a third part at a lower price (thus being a tricky contender of P). Notice, however, that evidence of U 's crime cannot be implemented in our calculus (also in real life there would be an external authority entitled to discover the crime and inform the publisher).

Type Soundness. We now adapt the static typing of Section 3.3.1 to the new scenario. First, notice that we do not need a specific capability to enable **revoke**: the operation is enabled only if l has previously delivered δ to l' , and this is checked at run-time, see rule (RED-REVOKE). Hence, the static typing system is modified adding the following rule

$$(\text{TD-REV}) \quad \frac{\Gamma \vdash_l^L P \triangleright \underline{P}}{\Gamma \vdash_l^L \mathbf{revoke}(\delta)@l'.P \triangleright \mathbf{revoke}(\delta)@l'.\underline{P}}$$

Moreover, rule (TD-NEW) now relies on \leq' .

Like for the previous variants, the typing can only rely on statically assigned capabilities; indeed, annotated capabilities can be revoked in unpredictable ways. Again, this forces us to also mark all those actions whose target is a variable because we cannot know if the action will be enabled by a revocable capability or not. This is why in rule (RED-EVAL''') we have used function $static(\cdot)$, that is also exploited to define well-typedness and executableness, as follows.

Definition 3.3.14 *A net is well-typed if, for each node $l ::^\gamma C$, there exists a component \underline{C} such that*

$$pol(static(\gamma)) \vdash_l C \triangleright \underline{C}$$

A net is executable if, for each node $l ::^\gamma C$, it holds that

$$pol(static(\gamma)) \vdash_l C \triangleright C$$

The definition of run-time errors now relies on the following variation of rule (ERRACT)

$$\frac{pol(\gamma)(tgt(a)) \not\sqsubseteq_{\Pi} \{cap(a)\}}{l ::^\gamma a.P \nearrow}$$

and soundness of the revised framework can be formulated and proved like in Theorem 3.3.12.

To conclude, notice that, here and in the other variants on capability loss, the soundness theorems can be essentially proved like in Section 3.3.2. This is due to the fact that, for the static typing, we only consider the capabilities always available, i.e. those capabilities that cannot be consumed, that never expires and that cannot be revoked. The marking mechanism, that does never give rise to run-time errors, is exploited whenever capabilities that can become unavailable are required.

3.3.5 Discussion on Capability Management

We now briefly discuss some language design issues related to the management of capabilities. Our type theory is largely independent from the underlying language. More specifically, it is possible to define a type theory similar to the one for μKLAIM we have presented in this section, whenever we have a language with the following features: (i) a set of (possibly remote) process operations and a set of corresponding capabilities, (ii) an ordering relation over capabilities, and (iii) some linguistic primitives for exchanging capabilities.

Notice that, however, several choices were possible when defining most of the functions used throughout this section; we shall only comment on some of them.

- When a tuple is produced, no check is made to control that the capabilities delivered through the tuple do agree with the access control policy of the node where the tuple is being inserted. Such ‘compatibility’ check is desirable if the point of view is taken that a node is responsible for the capabilities

it provides; clearly, this would give rise to a different notion of well-typed nets. The check could be implemented by appropriately modifying the check on grantings of Table 3.6 so that also the access control policy of the target node is taken into account.

- In rule (M₂) only the capabilities actually required by the process enrich the type of the node performing an action **in/read**. An alternative could be to enrich the type with *all* the capabilities delivered to the receiving node by the producer of the tuple.
- The definition of grantings used here is simpler than that of [80]. In *loc.cit.*, we can write $[l \mapsto \bar{\pi}]$ to mean that we want to pass to l all the capabilities currently owned, except for those in π . By taking $\pi = \emptyset$, it is thus possible to pass all the capabilities owned at run-time (that are statically unknown and unpredictable). Moreover, another way to make our theory more powerful is to add a reserved keyword **others** that could be used to denote all the nodes of a net; thus, the granting $[\mathbf{others} \mapsto \pi]$ would specify that the capabilities in π are granted to any node. Both these features can be easily accommodated in the theory presented in this section, but they require more run-time checks and more complicated definitions. Their integration in the framework we have presented is an easy task and it is left to the interested reader.

The theory presented in Section 3.3.4 is perhaps the simplest way to model capabilities revocation. We conclude by sketching more elaborated scenarios.

- According to rule (RED-REVOKE), the process $\mathbf{revoke}(\delta)@l'.P$ is stuck if only a subtype of δ is present in γ' . If we want to avoid this block, we can define the type $\gamma_1 \sqcap \gamma_2$ to be the greatest common subtype (w.r.t. \leq') of both γ_1 and γ_2 , and redefine (RED-REVOKE) to be

$$\frac{\gamma' = \gamma'', (\gamma' \sqcap \delta^{l'})}{l ::^\gamma \mathbf{revoke}(\delta)@l'.P \parallel l' ::^{\gamma'} C' \mapsto l ::^\gamma P \parallel l' ::^{\gamma''} C'}$$

Thus, we can remove from γ' the greatest subtype of δ delivered by l .

- The proposed formulation rules out direct attacks aimed at revoking as many capabilities as possible to reduce the functionality of a system. These attacks can be mounted by executing actions $\mathbf{revoke}(\delta)@l'$ by a process running at l , where l did not delivered δ to l' . However, one can easily imagine a scenario in which l spawns such a malicious process over an l'' that delivered δ to l' . A simple way to avoid this is to define two typing systems: the first one is \vdash_l , the other one, denoted by $\parallel \vdash_l$, is defined as the first one but without rule (TD-REV). We still use \vdash_l in the definitions of well-typed and of executable nets, while we use $\parallel \vdash_l$ in rule (RED-EVAL) (in this way we block incoming agents containing actions **revoke**). This solution can however be

over-restricting: a better (but more complex) solution is to define \Vdash in such a way that $\mathbf{revoke}(\delta)@l'$ is deemed legal only if it is syntactically preceded by an action **out** delivering l' some supertype of δ .

- The last scenario we consider is when l_1 delivers δ to l_2 and then l_2 delivers δ to l . Should it be legal for l_1 to perform an action **revoke** over l ? In the framework of Section 3.3.4 it is not. However, we could model this scenario by annotating capabilities with subsets of \mathcal{S} ; each such subset should represent all the (unordered) paths leading to the acquisition of the capability. E.g., if c is annotated with the set $\{\{l_1, l_2\}, \{l'_1, l'_2, l'_3\}\}$ in the annotated type of l , then c has been delivered to l through l_1 and l_2 and, independently, through l'_1 , l'_2 and l'_3 . Clearly, the semantics has to be modified to enable all the l_i s and l'_j s to perform actions **revoke** over l .

3.4 Fine-grained Controls on Process Activities

We now present a second evolution of the basic type system of Section 3.2; as we already said, it can be readily combined with the theory of Section 3.3 to yield a very powerful and flexible mean to control program executions. The type system we present in this section mainly implements two ideas leading to a fine-grained control:

1. the type system exploits the source of mobile processes for granting them different privileges, and
2. process capabilities regulate operations over parts of a tuple space (i.e., the multiset of all those tuples contained in the tuple space that can be described by a given pattern).

These desirable features can be found in real systems like, e.g., UNIX, where different users can have different privileges and different files can be manipulated by different allowed operations.

3.4.1 Fine-grained Types

We start by adapting capabilities and types of Definition 3.1.1 to the new scenario. We want to implement policies such that, for example, if l trusts l' , then l 's access control policy could accept processes coming from l' (that will be called l' -processes) and let them accessing any tuple in l 's TS. If l' is not totally trusted, then l 's access control policy could grant l' -processes the capabilities for executing **in/read** only over tuples that do not contain classified data, for example tuples starting with a field containing the value “*public*”.

$\frac{\mathcal{T}(p') \subseteq \mathcal{T}(p)}{\langle c, p \rangle \sqsubseteq_{\Pi} \langle c, p' \rangle}$	$\frac{\pi_1 \subseteq \pi_2}{\pi_2 \sqsubseteq_{\Pi} \pi_1}$	$\frac{\pi_1 \sqsubseteq_{\Pi} \pi'_1 \quad \pi_2 \sqsubseteq_{\Pi} \pi'_2}{\pi_1 \cup \pi_2 \sqsubseteq_{\Pi} \pi'_1 \cup \pi'_2}$
---	---	---

Table 3.12: Rules for Fine-grained Capability Ordering

Capabilities

Capabilities are used to specify the allowed process operations and are formally defined as

$$C \triangleq \{e, n\} \cup \{ \langle c, p \rangle : c \in \{i, r, o\} \wedge p \subseteq_{\text{fin}} \mathcal{P} \}$$

where $\mathcal{P} \triangleq (\mathcal{L} \cup \{\mathbf{from}, -\})^+$ is the set of all *patterns*. Capabilities e and n enable process migration and node creation, like before. A capability of the form $\langle c, p \rangle$ enables the operation whose name's first character is c (i.e. **in** if c is i , and so on); operation arguments must comply with the finite set of patterns p if $p \neq \emptyset$, and are not restricted otherwise (in this case, we write c instead of $\langle c, \emptyset \rangle$). Like tuples and templates, *patterns* are finite, not empty sequences of fields; *pattern fields* may be localities, the reserved word **from** (denoting the last locality visited by a mobile process) and the “don't care” symbol $-$ (denoting any template field). Thus, for instance, the capability $\langle i, \{(\text{"public"}, -), (3, -, \mathbf{from})\} \rangle$ enables the operations **in**(“public”, ! x)@... and **in**(3, ! $x : \pi, l$)@... for a l -process, while disables operations like **in**(“private”, ! x)@... .

We use π to denote a non-empty subset of C such that, if $\langle c, p \rangle \in \pi$ and $\langle c', p' \rangle \in \pi$, then $c \neq c'$. Π will denote the set of all these π s.

We say that a template *complies with* a pattern if the template is obtained by replacing in the pattern all occurrences of **from** with a locality, and any occurrence of $-$ with any template field allowed by the syntax. Given a non-empty set of patterns p , we write $\mathcal{T}(p)$ to denote the set of all templates complying with patterns in p . By definition, $\mathcal{T}(\emptyset)$ denotes the set of all templates. Since tuples are also templates (see Table 2.4), the previous definitions also apply to tuples.³

We now extend the ordering between capabilities given in Definition 3.1.2; formally, it is the least reflexive and transitive relation induced by the rules in Table 3.12. The chosen ordering relies the assumption that, if a process is allowed to perform a **read/in/out** over arguments complying with patterns in p , then it is allowed to perform the same operation over arguments complying with any set of patterns p' that has at most the same ‘complying templates’ as p .

³Notice that the definition of pattern fields affects, via the relation ‘complies with’, the ability of our types to control the tuples accessed by process operations. However, our framework is largely independent of the choice of a specific set of fields. For instance, adding to μKLAIM basic values (of type `int`, `string`, `bool`, ...), we could also permit fields of the form \neg_{Θ} , for any type Θ of basic values. Then, the relation ‘complies with’ could be defined in such a way that an occurrence of \neg_{Θ} would be replaced by any value/variable of type Θ . In this way, a finer control could be exercised on the tuples accessed by processes because we could distinguish, e.g., between a tuple field containing an integer and one containing a string.

Types

Types, ranged over by Δ , are functions of the form

$$\Delta : \mathcal{L} \cup \{\mathbf{any}\} \rightarrow_{\text{fin}} (\{\perp\} \cup (\mathcal{L} \cup \{\mathbf{any, from}\} \rightarrow_{\text{fin}} \Pi))$$

where \rightarrow_{fin} means that the function maps only a finite subset of its domain to significant values (i.e. values different from \perp and \emptyset). With abuse of notation, we use \perp to also denote the empty type, i.e. the function mapping any element of its domain to \perp . Moreover, by letting λ to range over $\mathcal{L} \cup \{\mathbf{any, from}\}$, we shall write any Δ different from \perp as a non-empty list $[\lambda_i \mapsto [\lambda_{ij} \mapsto \pi_{ij}]_{j=1, \dots, k_i}]_{i=1, \dots, n}$; in this case, we shall denote the set $\{\lambda_i\}_{i=1, \dots, n}$ as $\text{dom}(\Delta)$.

Like before, types express the access control policies of nodes. Intuitively, if the type Δ of a node with address l contains the element $[l' \mapsto l'' \mapsto \pi]$, then l' -processes located at l are allowed to perform over l'' only the operations enabled by π . The reserved word **any** is used to refer any node of the net. If it occurs in the domain of Δ then it collects the capabilities granted to processes coming from any node of the net (i.e. $[\mathbf{any} \mapsto l'' \mapsto \pi]$ grants all processes the capabilities π over l''). If **any** is contained in the domain of $\Delta(l')$, for some l' , then it is used for denoting the operations that l' -processes located at l are allowed to perform over any node of the net (i.e. $[l' \mapsto \mathbf{any} \mapsto \pi]$ grants l' -processes the capabilities π over all net nodes). The reserved word **from** stands for the last node visited by a process and is used to grant capabilities over this node whatever it is; thus, for instance, $[\mathbf{any} \mapsto \mathbf{from} \mapsto \pi]$ grants l' -process spawned at l the capabilities π over l' . The type \perp expresses total absence of capabilities.

We now revise the notion of *subtyping*; again, this notion is derived from the standard preorder over functions, by also using the pointwise union of functions, denoted by \uplus .

Definition 3.4.1 *The subtyping relation, \leq , is the least reflexive and transitive relation closed under the rule*

$$\frac{\forall \lambda \in \text{dom}(\Delta_1) : \Delta_1(\lambda) \leq \Delta_2(\lambda) \uplus \Delta_2(\mathbf{any})}{\Delta_1 \leq \Delta_2}$$

where \leq extends the relation of Definition 3.1.3 to take into account associations for the reserved name **any**. Formally, it is the least reflexive and transitive relation closed under the following rules:

$$\frac{m \leq n \quad \forall i = 1, \dots, m : \pi'_i \cup \pi'_k \sqsubseteq_{\Pi} \pi_i}{[\lambda_i \mapsto \pi_i]_{i=1, \dots, m} \leq [\lambda_j \mapsto \pi'_j]_{j=1, \dots, n} \quad \lambda_k = \mathbf{any}}$$

$$\frac{m \leq n \quad \forall i = 1, \dots, m : \pi'_i \sqsubseteq_{\Pi} \pi_i}{[\lambda_i \mapsto \pi_i]_{i=1, \dots, m} \leq [\lambda_j \mapsto \pi'_j]_{j=1, \dots, n} \quad \forall j. \lambda_j \neq \mathbf{any}}$$

Notice that $\perp \leq \Delta$ for any Δ , since $\text{dom}(\perp) = \emptyset$.

We finally introduce the notion of *well-formed* types, that will be useful when proving soundness of our system.

Definition 3.4.2 *The type Δ is l -well-formed whenever the following conditions hold:*

1. If **from** $\in \text{dom}(\Delta(\lambda))$ then $\lambda = \mathbf{any}$
2. For each $\lambda \in \text{dom}(\Delta)$, it holds that $\Delta(\lambda) \leq \Delta(l)$

Apart from technical reasons, this definition seems reasonable when considering Δ to be the type of locality l . Indeed, the first condition is not too restrictive, because the use of **from** is really necessary only when no knowledge of the last node visited by processes is available (i.e. when using **any**). The second condition says that l grants to λ -processes (for $\lambda \in \text{dom}(\Delta)$) no more capabilities than those granted to its local processes, i.e. those processes statically allocated at l . This seems reasonable because usually local code is more trusted than foreign code and, hence, it is assigned more capabilities.

3.4.2 Static Semantics

For each node of a net, say $l ::^\Delta C$, the static type checker analyses the operations that the component C located at l intends to perform and determines whether they are enabled by the access policy Δ or not. A type context Γ is now a function of the form

$$\mathcal{L} \cup \{\mathbf{any}\} \rightarrow_{\text{fin}} \Pi$$

The updating of a type context Γ with the type annotations specified within a template T is still denoted $\text{upd}(\Gamma, T)$ and is formally defined like in the previous sections.

Type judgements for processes take the form $\Gamma \vdash_l^\Delta P$ and, like before, state that, within the context Γ , P can be safely executed once located at l , whose policy is Δ . Type judgements are inferred by using the rules in Table 3.13 that should be now quite explicative. For operations **out**, **in**, **read** and **eval**, the inference requires the capability associated to the operation to be enabled by the capabilities owned over the target u or over all the net sites. Instead, for operation **new**, the capability n must be owned by the site l executing the operation. In this case, it is assumed that the creating node owns over the created one all the capabilities it owns on itself. Moreover, the type Δ' , declared as the policy of the new node, must be well-formed for l' and less permissive than its creator's type Δ . To this aim, we also need to add the associations contained in Γ to Δ when checking relation \leq , since Δ' is allowed to contain names that are in the domain of Γ but not in the domain of Δ : consider for example the case of the process

$$\mathbf{in}(!x : \pi)@l.\mathbf{new}(l' : [l' \mapsto x \mapsto \pi']).\dots$$

$\frac{}{\Gamma \vdash_l^\Delta \mathbf{nil}}$	$\frac{}{\Gamma \vdash_l^\Delta \langle t \rangle}$
$\frac{}{\Gamma \vdash_l^\Delta X}$	$\frac{\Gamma \vdash_l^\Delta P}{\Gamma \vdash_l^\Delta \mathbf{rec} X.P}$
$\frac{\Gamma \vdash_l^\Delta C_1 \quad \Gamma \vdash_l^\Delta C_2}{\Gamma \vdash_l^\Delta C_1 \mid C_2}$	$\frac{\Gamma(u) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \{e\} \quad \Gamma \vdash_l^\Delta P}{\Gamma \vdash_l^\Delta \mathbf{eval}(Q)@u.P}$
$\frac{\Gamma(u) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle o, p \rangle \quad t \in \mathcal{T}(p) \quad \Gamma \vdash_l^\Delta P}{\Gamma \vdash_l^\Delta \mathbf{out}(t)@u.P}$	
$\frac{\Gamma(u) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle i, p \rangle \quad T \in \mathcal{T}(p) \quad \mathit{upd}(\Gamma, T) \vdash_l^\Delta P}{\Gamma \vdash_l^\Delta \mathbf{in}(T)@u.P}$	
$\frac{\Gamma(u) \cup \Gamma(\mathbf{any}) \sqsubseteq_{\Pi} \langle r, p \rangle \quad T \in \mathcal{T}(p) \quad \mathit{upd}(\Gamma, T) \vdash_l^\Delta P}{\Gamma \vdash_l^\Delta \mathbf{read}(T)@u.P}$	
$\frac{\Gamma(l) \sqsubseteq_{\Pi} \{n\} \quad \Gamma \cup [l' \mapsto \Gamma(l)] \vdash_l^{\Delta \cup [l \mapsto l' \mapsto \Delta(l)(l)]} P \quad \Delta \text{ is } l'\text{-well-formed} \quad \Delta' \leq \Delta \cup [l \mapsto ([l' \mapsto \Delta(l)(l)] \cup \Gamma)]}{\Gamma \vdash_l^\Delta \mathbf{new}(l' : \Delta').P}$	

Table 3.13: Inference Rules for the Fine-grained System

We conclude this section by defining *well-typed* nets w.r.t. the fine-grained type system.

Definition 3.4.3 *A net N is well-typed if for each node $l ::^\Delta C$ in N it holds that Δ is l -well-formed and $\Delta(l) \vdash_l^\Delta C$.*

3.4.3 Dynamic Semantics and Type Soundness

Like in the previous sections, the operational semantics of Table 2.5 needs to be modified by adding some runtime type checks. The structural congruence is readily

$match_{\Delta(l)}(l', l') = \epsilon$	$\frac{\Delta(l)(l') \cup \Delta(l)(\mathbf{any}) \sqsubseteq_{\Pi} \pi}{match_{\Delta(l)}(!u : \pi, l') = [l'/u]}$
$match_{\Delta(l)}(T_1, t_1) = \sigma_1$	$match_{\Delta(l)}(T_2, t_2) = \sigma_2$
$match_{\Delta(l)}(T_1, T_2, t_1, t_2) = \sigma_1 \circ \sigma_2$	

Table 3.14: Matching Rules with Fine-grained Types

adapted from Section 3.2.2. With respect to Table 3.3, the main modifications affect the pattern-matching predicate and the rules for actions **eval** and **new**. The revised definition for pattern-matching readily extends the definition given in Table 3.2; for the sake of clarity, its definition is given in Table 3.14. The main novelty is that, when trying to replace a formal field $!u : \pi$ with a value l' , the capabilities owned by l' 's static code over any node can be added to the capabilities owned over l' to obtain the needed capabilities in π .

The μKLAIM operational semantics with fine-grained types is the least relation induced by the rules in Table 3.15. Let us comment on the key modifications. Rule (RED-EVAL) says that the migrating process Q must be checked against the union of the capabilities that the access control policy Δ' of the target node l' assigns to processes coming from l and to processes coming from any node (in this last case, occurrences of **from** must be interpreted as l , as stated by the syntactic substitution $\Delta'(\mathbf{any})[l/\text{from}]$ of **from** with l in function $\Delta'(\mathbf{any})$).

To conclude, notice that Proposition 3.2.3 still holds in this framework.

Type Soundness.

We can now adapt the results of Section 3.2.3 to the setting of fine-grained types. The formulations of Lemmas 3.2.5 and 3.2.6 need to be slightly modified and their proofs is carried-on like in Section 3.2.3; Lemma 3.2.7 is stated in the same way and proved similarly.

Lemma 3.4.4 (Weakening) *If $\Gamma \vdash_l^\Delta C$ then $\Gamma' \vdash_l^\Delta C$, for every Γ' such that $\Gamma \preceq \Gamma'$.*

Lemma 3.4.5 (Substitutivity)

1. *If $\Gamma \vdash_l^\Delta C$ then, for any substitution σ , $\Gamma\sigma \vdash_l^\Delta C\sigma$.*
2. *If $\Gamma \vdash_l^\Delta C$ and $\Gamma \vdash_l^\Delta Q$, then $\Gamma \vdash_l^\Delta C[Q/X]$.*

Theorem 3.4.6 (Subject Reduction) *If N is well-typed and $N \mapsto N'$, then N' is well-typed.*

Proof: The proof is a smooth adaption of the corresponding one of Theorem 3.2.8; we only discuss the two most significant differences.

<p>(RED-OUT)</p> $\frac{}{l ::^\Delta \mathbf{out}(t)@l'.P \parallel l' ::^{\Delta'} \mathbf{nil} \mapsto l ::^\Delta P \parallel l' ::^{\Delta'} \langle t \rangle}$
<p>(RED-EVAL)</p> $\frac{\Delta'(l) \uplus (\Delta'(\mathbf{any})[l]_{\text{from}}) \vdash_{l'}^{\Delta'} Q}{l ::^\Delta \mathbf{eval}(Q)@l'.P \parallel l' ::^{\Delta'} \mathbf{nil} \mapsto l ::^\Delta P \parallel l' ::^{\Delta'} Q}$
<p>(RED-IN)</p> $\frac{\mathit{match}_{\Delta(l)}(T, t) = \sigma}{l ::^\Delta \mathbf{in}(T)@l'.P \parallel l' ::^{\Delta'} \langle t \rangle \mapsto l ::^\Delta P\sigma \parallel l' ::^{\Delta'} \mathbf{nil}}$
<p>(RED-READ)</p> $\frac{\mathit{match}_{\Delta(l)}(T, t) = \sigma}{l ::^\Delta \mathbf{read}(T)@l'.P \parallel l' ::^{\Delta'} \langle t \rangle \mapsto l ::^\Delta P\sigma \parallel l' ::^{\Delta'} \langle t \rangle}$
<p>(RED-NEW)</p> $\frac{}{l ::^\Delta \mathbf{new}(l' : \Delta').P \mapsto (\nu l')(l ::^\Delta \uplus [l \mapsto l' \mapsto \Delta(l)(l)] P \parallel l' ::^{\Delta'} \mathbf{nil})}$
<p>(RED-SPLIT)</p> $\frac{l ::^\Delta C_1 \parallel l ::^\Delta C_2 \parallel N \mapsto (\widetilde{\nu l})(l ::^\Delta C'_1 \parallel l ::^\Delta C'_2 \parallel N')}{l ::^\Delta C_1 C_2 \parallel N \mapsto (\widetilde{\nu l})(l ::^\Delta C'_1 C'_2 \parallel N')}$
<p>with rules (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 2.3</p>

Table 3.15: Operational Semantics with Fine-grained Types

(RED-EVAL). Well-typedness of node $l ::^\Delta P$ is inferred like in the previous case. We are only left to prove that $\Delta'(l') \vdash_{l'} Q$ since, by hypothesis, $\Delta'(l') \vdash_{l'} P'$. By the premise of rule (RED-EVAL), we have that $\Delta'(l) \uplus (\Delta'(\mathbf{any})[l]_{\text{from}}) \vdash_{l'} Q$. If we prove that

$$\Delta'(l) \uplus (\Delta'(\mathbf{any})[l]_{\text{from}}) \leq \Delta'(l')$$

then we can conclude by using Lemma 3.4.4. Indeed, it suffices to prove that both $\Delta'(l)$ and $\Delta'(\mathbf{any})[l]_{\text{from}}$ are subtypes of $\Delta'(l')$, since trivially the pointwise extension of partial functions respects this property. By l' -well-formedness of type Δ' , we have that:

- $\Delta'(l) \leq \Delta'(l')$ (by condition 2. of Definition 3.4.2)
- let $\lambda \in \text{dom}(\Delta'(\mathbf{any})[l]_{\text{from}})$; then
 - if $\lambda \neq l$ then $(\Delta'(\mathbf{any})[l]_{\text{from}})(\lambda) = \Delta'(\mathbf{any})(\lambda)$ and, since

- $\Delta'(\mathbf{any}) \leq \Delta'(l')$ (again, by Definition 3.4.2(2)), it holds that $\Delta'(l')(\lambda) \sqsubseteq_{\Pi} (\Delta'(\mathbf{any})[l'_{\text{from}}])(\lambda)$
- otherwise, it is easy to check that $(\Delta'(\mathbf{any})[l'_{\text{from}}])(l)$ is $\Delta'(\mathbf{any})(l) \cup \Delta'(\mathbf{any})(\mathbf{from})$. By Definition 3.4.2(2), we have that $\Delta'(l')(l) \sqsubseteq_{\Pi} \Delta'(\mathbf{any})(l)$. By Definition 3.4.2(1), we also know that $\mathbf{from} \notin \text{dom}(\Delta'(l'))$; hence, because $\Delta'(\mathbf{any}) \leq \Delta'(l')$ and by Definition 3.4.1, it must be that $\Delta'(l')(\mathbf{any}) \sqsubseteq_{\Pi} \Delta'(\mathbf{any})(l)$.

These facts amount to say that $\Delta'(\mathbf{any})[l'_{\text{from}}] \leq \Delta'(l')$.

(RED-NEW). First of all, we have to prove that the types occurring in the resulting net are still well-formed. Type $\Delta \uplus [l \mapsto l' \mapsto (\Delta(l)(l) - \{n\})]$ is l -well-formed, since it is obtained from the l -well-formed type Δ by adding capabilities to l -processes; type Δ' is verified to be l' -well-formed by the static checker. Then, the fact that $l ::^{\Delta \uplus [l \mapsto l' \mapsto \Delta(l)(l)]} P$ is well-typed is proved similarly to the case for (RED-NEW) in Theorem 3.2.8; we leave the details to the reader. ■

The notion of run-time errors is adapted from Table 3.4 by upgrading rule (ERRACT) as follows

$$\frac{\Delta(l)(\text{tgt}(a)) \cup \Delta(l)(\mathbf{any}) \not\sqsubseteq_{\Pi} \{\text{cap}(a)\}}{l ::^{\Delta} a.P \nearrow}$$

Theorem 3.4.7 (Type Safety) *If N is well-typed then $N \nearrow$ does not hold.*

Proof: Like in Theorem 3.2.10, we proceed by contradiction. The only relevant difference is for the base case, where we have to reason on two sub-cases:

- $a.P$ was part of a l' -process migrated to l . In this case, it cannot be $\Delta(l') \uplus (\Delta(\mathbf{any})[l'_{\text{from}}]) \vdash_l^{\Delta} a.P$, otherwise, by Lemma 3.4.4, $\Delta(l) \vdash_l^{\Delta} a.P$ and in particular $\Delta(l)(\text{tgt}(a)) \cup \Delta(l)(\mathbf{any}) \sqsubseteq_{\Pi} \{\text{cap}(a)\}$. But then, the premise of rule (RED-EVAL) in Table 3.15 would not be satisfied and hence the migration of the l' -process containing $a.P$ would have been blocked. This contradicts the assumption on the origin of process $a.P$.
- $a.P$ was located at l at the outset. In this case, it must be that $\Delta(l)(\text{tgt}(a)) \cup \Delta(l)(\mathbf{any}) \not\sqsubseteq_{\Pi} \{\text{cap}(a)\}$, thus falsifying the premise of the static inference rule for action a . This contradicts the well-typedness hypothesis. ■

Corollary 3.4.8 (Global Type Soundness) *If N is well-typed and $N \mapsto^* N'$, then $N \nearrow$ does not hold.*

Theorem 3.4.9 (Local Type Soundness) *Let N be a net. If N_S is well-typed and $N \mapsto^* N'$, then $N'_S \nearrow$ does not hold.*

3.4.4 Example: A Bank Account Management System

In this section, we use our approach to model the simplified behaviour of a bank account management system. For ensuring compliance with the access control policy of the bank some aspects of our setting, such as the possibility of granting different privileges to processes coming from different source nodes and the dynamic type checking of mobile processes when they migrate, have proved to be crucial.

We suppose that a bank is located at a node with address l_B and can receive and manage requests coming from many users located at nodes with addresses $l_U, l_{U'}, \dots$. The bank must provide the users with typical account managing operations: opening/closing accounts, putting/getting money in/from accounts, and making statements of accounts. For simplicity, we shall omit some details and technical operations that in reality take place, like, e.g., charging taxes and dealing with improper operations (like the attempt of getting more money than that really available).

For permitting the bank to check the operations that users intend to perform, we assume that users cannot perform remote operations over l_B except for sending processes. This can be achieved by using the type system presented in Section 3.3: the bank can pass its address, that should be fresh, only associated with the e capability. Hence, if a user U wants to require an operation to the bank, it can only send a process to l_B (thus virtually moving to the bank) which will interact locally with the proper operation handler. The user process, once it has been accepted (i.e. after its compliance with the bank access control policy has been checked), can require the operation by locally producing a tuple whose first field contains the name of the operation and whose second field contains the address of the user node (used to identify the user that made the request). Depending on the operation, the tuple could have other fields containing the amount of money involved in the operation and the account number receiving the money.

The node implementing the bank is illustrated in Table 3.16. First, the bank creates a new node that will contain its clients accounts, stored as tuples of the form $(userAddress, amount)$. This node acts just as a repository for tuples and will not be used for spawning processes, thus it has assigned the empty type \perp . Then, five different handler processes, one for each kind of operation, are concurrently spawned. Each handler continuously waits for a request. When such a request arrives, the proper handler executes its task by remotely accessing the reserved locality and then reports locally a confirmation of action completion. The client process performing the request waits for such a confirmation and then brings it back to its original locality. This last operation is performed by means of a migration thus providing the user node with the chance of controlling the operation.

Notice that, by taking advantage of the semantics of μ KLAIM operations, the simple handlers of Table 3.16 implement the mutual exclusion needed to ensure the correctness of concurrent operations over shared data. Indeed, once a handler H has withdrawn the tuple representing an account (i.e. once H has locked the account), the other handlers cannot perform other tasks on the same account until

$l_B ::^{\Delta_B} \mathbf{new}(l_S : \perp).(OpenH(l_S) PutH(l_S) GetH(l_S) ReadH(l_S) CloseH(l_S))$	
<p>where:</p>	
$OpenH(u) \triangleq \mathbf{rec} X.\mathbf{in}(\text{"open"}, !x, !y)@l_B.$	$(X \mathbf{out}(x, y)@u.\mathbf{out}(\text{"OKopen"}, x, y)@l_B)$
$PutH(u) \triangleq \mathbf{rec} X.\mathbf{in}(\text{"put"}, !x, !y, !w)@l_B.$	$(X \mathbf{in}(w, !z)@u.\mathbf{out}(w, z + y)@u.\mathbf{out}(\text{"OKput"}, x, y, w)@l_B)$
$GetH(u) \triangleq \mathbf{rec} X.\mathbf{in}(\text{"get"}, !x, !y)@l_B.$	$(X \mathbf{in}(x, !z)@u.\mathbf{out}(x, z - y)@u.\mathbf{out}(\text{"OKget"}, x, y)@l_B)$
$ReadH(u) \triangleq \mathbf{rec} X.\mathbf{in}(\text{"read"}, !x)@l_B.$	$(X \mathbf{read}(x, !y)@u.\mathbf{out}(\text{"OKread"}, x, y)@l_B)$
$CloseH(u) \triangleq \mathbf{rec} X.\mathbf{in}(\text{"close"}, !x)@l_B.$	$(X \mathbf{in}(x, !y)@u.\mathbf{out}(\text{"OKclose"}, x, y)@l_B)$
$\Delta_B \triangleq$	$[l_B \mapsto [l_B \mapsto \{i, o, r, n\},$
	$\mathbf{any} \mapsto \{e\}],$
	$\mathbf{any} \mapsto [\mathbf{from} \mapsto \{e\},$
	$l_B \mapsto \{ \langle o, \{$
	$(\text{"open"}, \mathbf{from}, -),$
	$(\text{"put"}, \mathbf{from}, -, -),$
	$(\text{"get"}, \mathbf{from}, -),$
	$(\text{"read"}, \mathbf{from}),$
	$(\text{"close"}, \mathbf{from})$
	$\} \rangle,$
	$\langle i, \{$
	$(\text{"OKopen"}, \mathbf{from}, -),$
	$(\text{"OKput"}, \mathbf{from}, -, -),$
	$(\text{"OKget"}, \mathbf{from}, -),$
	$(\text{"OKread"}, \mathbf{from}, -),$
	$(\text{"OKclose"}, \mathbf{from}, -)$
	$\} \rangle$
	$]$
	$]$

Table 3.16: The node implementing the bank

H writes the updated tuple (i.e. H releases the lock).

The access control policy Δ_B is so defined that ‘sensible’ operations over the accounts of a user U (like getting some money and reading/closing the account) can only be requested by l_U -processes, while operations like putting some money can be requested by processes coming from any node. Moreover, the only remote operation processes are allowed to perform is to come back to their source site. Therefore, a l_U -process can request to the bank sensible operations only over U ’s accounts and can deliver the confirmations only to l_U . Typical processes acting on behalf of a user U are illustrated in Table 3.17, where the parameter s denotes an

$OpenR(s) \triangleq$	$\mathbf{eval}(\mathbf{out}(\text{"open"}, l_U, s)@l_B.\mathbf{in}(\text{"OKopen"}, l_U, s)@l_B.\mathbf{eval}(\mathbf{out}(\text{"OKopen"}, s)@l_U)@l_U)@l_B$
$PutR(s, l_{U'}) \triangleq$	$\mathbf{eval}(\mathbf{out}(\text{"put"}, l_U, s, l_{U'})@l_B.\mathbf{in}(\text{"OKput"}, l_U, s, l_{U'})@l_B.\mathbf{eval}(\mathbf{out}(\text{"OKput"}, s, l_{U'})@l_U)@l_U)@l_B$
$GetR(s) \triangleq$	$\mathbf{eval}(\mathbf{out}(\text{"get"}, l_U, s)@l_B.\mathbf{in}(\text{"OKget"}, l_U, s)@l_B.\mathbf{eval}(\mathbf{out}(\text{"OKget"}, s)@l_U)@l_U)@l_B$
$ReadR \triangleq$	$\mathbf{eval}(\mathbf{out}(\text{"read"}, l_U)@l_B.\mathbf{in}(\text{"OKread"}, l_U, !x)@l_B.\mathbf{eval}(\mathbf{out}(\text{"OKread"}, x)@l_U)@l_U)@l_B$
$CloseR \triangleq$	$\mathbf{eval}(\mathbf{out}(\text{"close"}, l_U)@l_B.\mathbf{in}(\text{"OKclose"}, l_U, !x)@l_B.\mathbf{eval}(\mathbf{out}(\text{"OKclose"}, x)@l_U)@l_U)@l_B$

Table 3.17: Processes of a user U requesting bank operations

amount of money and the parameter $l_{U'}$ denotes an account.

The only possibility for a malicious node to illegally access U 's accounts is to pass through l_U , using a process like $\mathbf{eval}(\mathbf{eval}(\text{MaliciousReq})@l_B)@l_U$. Hence, U has to protect itself from these attacks by granting an e capability over l_B only to processes coming from totally trusted nodes: the access control policy of l_U must contain the element $[l \mapsto l_B \mapsto \{e\}]$ only if U trusts the user located at l . However, U can trust l only if U trusts all l' trusted by l (in fact, a node trusted by l can send to l a process that is then allowed to spawn a process at U containing requests on U 's accounts). Notice that we do not aim at modelling here sophisticated trust managements; a wide literature on this topic is available and can be accommodated in our setting.

Finally, notice that only the handler processes can access the node dynamically created whose address is l_S . Indeed, when such node is created, the operational semantics dynamically extends Δ_B with $[l_B \mapsto l_S \mapsto \{i, r, o\}]$ thus enabling all, and only those, processes initially allocated at l_B to perform **in/read/out** operations over l_S .

3.5 Related Work

There is a lot of work on type systems for security in calculi with process distribution and mobility. We conclude by surveying more strictly related work and by contrasting it against the theory we have presented in this chapter.

A lot of type systems for GCs are strictly related to security; among the others, we mention those for controlling

- the *types of the values exchanged* in communications [39, 25, 90],
- *Ambients mobility and ability to be opened* [36, 37, 105, 69, 43],
- resource access via policies for *mandatory and discretionary access control* [26, 27],

- *co-location* of all processes that intend to access a given channel [139, 153],
- the *effect of transmitted process abstractions* over local channels [154, 87].

The research line closest to ours is that on the $D\pi$ -calculus [90, 88, 129], where type systems control capabilities of mobile processes over located resources (i.e. communication channels). Differently from μKLAIM , node types describe permissions to use local channels (this is in sharp contrast with μKLAIM types that aim at controlling the remote operations that a network node can perform over the other network nodes). For establishing well-typedness, the type system in [90] needs considering the whole net, while that in [88] only needs local information (because processes are dynamically checked whenever they migrate), and thus is similar to our approach. In [129], the global knowledge about the net is split into several parts (that are updated independently, but still in accordance with the global knowledge, via dynamic knowledge acquisition through communication) and, to reduce the amount of dynamic controls, a relation of trust among nodes is exploited (thus, no process coming from a trusted node is type checked).

[153] presents $D\pi\lambda$, a process calculus that results from the integration of the call-by-value λ -calculus and the π -calculus, together with primitives for process distribution and remote process creation. Apart from the higher order and channel-based communication, the main difference with μKLAIM is that $D\pi\lambda$ localities are anonymous (i.e. not explicitly referrable by processes) and simply used to express process distribution. $D\pi\lambda$ comes equipped with a type system that controls how processes use resources (i.e. channels) by guaranteeing that in well-typed systems processes that intend to perform inputs at a given channel are co-located. In [154], a fine-grained type system for $D\pi\lambda$ is defined that permits controlling the effect of transmitted process abstractions (parameterised with respect to channel names) over local channels. Processes are assigned fine-grained types that, like interfaces, record the channels to which processes have access together with the corresponding capabilities, and process abstractions are assigned dependent functional types that abstract from channel names and types. This use of types is akin to μKLAIM 's one, though the differences between the underlying languages still remain.

Recently [87], the $D\pi$ has been extended to allow high-order data to be transmitted. In the resulting formalism, called *SafeDpi*, parameterised code may be sent between sites using higher-order channels. A host location may protect it self by only accepting code which conforms to a given type associated to the incoming port. A sophisticated static type system is then defined for these ports to restrict the capabilities and access rights of any process launched by incoming code. Dependent and existential types are used to add flexibility, allowing the behaviour of these launched processes to depend on the host's instantiation of the incoming code. However, similarly to [90, 154], the approach strongly relies on a global typing environment to type the whole net. In our view, this is a too demanding request in a global computing framework.

Finally, we want to mention some proposals for the Mobile Ambients calculus and its variants, albeit their network models and mobility mechanisms are very dif-

ferent from those of μKLAIM . As an example, we want to mention some differences between the mobility control in Ambient and in μKLAIM deriving from their different programming choices. Mobility in μKLAIM is, using Ambient terminology, *objective* (in that the moving process is spawned by an external process), while most of the work on Ambient-derived calculi uses *subjective* moves (explicitly programmed in the moving entities). The mobility control defined in [105] for the *Safe Ambient Calculus* (similar to that presented in [38] for the Mobile Ambient calculus) consists in identifying those ambients, called *immobile*, that cannot be opened and autonomously move, but can host local message exchanges and can receive incoming ambients (that, however, cannot be opened therein). A somehow similar behaviour can be implemented in μKLAIM by setting the policy of a node l to be $[l \mapsto \{r, i, o, e, n\}]$; however, this type cannot rule out remote communications using l as the target node since l 's type cannot restrict actions remotely performed. [69] extends [105] by introducing ambient *security levels*: the types used there rule out the movement through (and the opening of) ambients of lower security levels. We believe that this feature can be easily introduced in the μKLAIM setting.

Chapter 4

Types for Confining Data and Processes

The major goal of language-based security is the design of languages that are flexible, expressive and safe. Unfortunately, these are often contrasting requirements. For example, mobile code deeply increases flexibility and thus expressiveness of programming languages, but introduces new security problems related to unwanted accesses to classified data. Standard security issues (like, e.g., *secrecy* and *integrity* of data and program code) are complicated by the presence of code mobility: the use of cryptography, that is one of the most diffused techniques for ensuring security in distributed systems, must be strongly restricted in order to be safe. Indeed, malicious nodes can attack a mobile process and compromise its integrity through code modification or its secrecy through leakage of sensitive data. Similarly, malicious mobile processes might attempt to access or forge private data of the network nodes hosting them. Hence, we can hardly imagine to use mobile processes carrying confidential data (e.g. private keys) with them, or host nodes with classified information accessible to all incoming processes (whatever their source node be). Hence, the use of security mechanisms that back up and supplement cryptographic mechanisms becomes a major issue when developing systems of distributed and mobile processes where the compliance with some security policies must be guaranteed.

In this chapter, we define an approach that permits protecting the secrecy of both data residing on hosting nodes and of data carried by mobile processes, by relying on program annotation. Our approach is inspired by *Confined- λ* [98] and relies on annotating data with sets of node addresses, called *regions*, that specify the network nodes that can interact with them. Also nodes may have annotations that specify which nodes can send data and spawn processes to them. *Data annotations* enable programmers to control the set of nodes that can share specific data, and permit shading them from other nodes. *Node annotations*, instead, enable node administrators to control the set of data and processes each node can host; thus, the node can refuse malicious processes and unwanted data.

The language semantics is then designed to guarantee that computations proceed while respecting the region constraints. For example, a process P can access a datum d only if P 's execution does not export d outside its data region, say r , i.e. if P only writes d in network nodes included in r or, similarly, if P only carries d while migrating to nodes included in r . Enforcing similar constraints requires a form of code inspection that is performed, as much as possible, statically. This relieves the runtime semantics of the burden to make expensive checks and, then, improves efficiency.

The approach we shall present here is largely independent of a specific model: indeed, in [61] we adapted our approach to other two paradigmatic calculi for GC, namely $D\pi$ [90] and Mobile Ambients Calculus [41].

4.1 Controlling Data Movement via Types

We now set up a machinery based on typing that helps in protecting exchanged and local data in global computing applications. To this aim, we suggest annotating data with sets of network addresses, describing the sub-net where data can be used; these sets will be called *regions* and are formally defined as follows.

Definition 4.1.1 (Regions) *Regions, r , are either finite subsets of names or the distinct element \top , used to refer to the whole net. The set of all regions is \mathcal{R} ; it is partially ordered by the subset inclusion relation \subseteq and has \top as top element.*

The syntax of Table 2.4 is changed as follows:

$$\begin{aligned} N &::= \dots \mid l_{r_d::r_p} C \mid \dots \\ t &::= [u]_r \mid \dots \end{aligned}$$

The annotations allow programmers to fix the nodes that can share a given datum, and to avoid that the datum is accessed by untrusted processes (from untrusted nodes). Also network nodes are annotated with regions that specify the nodes that can send data and those that can spawn processes to them. Thus, nodes are annotated with two regions, say r_d and r_p . We should have $r_p \subseteq r_d$ since accepting processes is, in general, more dangerous than accepting data; however, no restriction on the model is imposed to deal with this issue. We shall assume that absence of region annotations stands for \top .

The language semantics guarantees that computations proceed according to region constraints. This property, that we call *safety*, can be intuitively stated as ‘‘A net N is safe if, for any datum d occurring in N associated to region r and for all possible evolutions of N , it holds that d will only cross and reside at nodes whose addresses are in r ’’.

To better understand the properties we want to model and the impact of our approach on system security, we present a simple client/server example. Suppose that a client C requires a service to a server S . Once S has verified the credentials of

C (e.g. its identity or its credit card information), it sends back a secret password, that C can change. C could then access the service by using the last set password. This protocol can be modelled by assuming two network addresses, l_C and l_S , hosting the processes P_C and P_S , respectively, defined as

$$\begin{aligned}
P_C &\triangleq \mathbf{out}(l_C, [cc_info]_{\{l_C, l_S\}}) @ l_S . \mathbf{in}(!y) @ l_C . \\
&\quad < \text{modify the password } y \text{ and access the service } > \\
P_S &\triangleq \mathbf{rec } X . \mathbf{in}(!x_1, !x_2) @ l_S . (X \mid < \text{check the credit card info } x_2 > . \\
&\quad \mathbf{new}(pwd) . \mathbf{out}([pwd]_{\{x_1, l_S\}}) @ x_1 . \\
&\quad < \text{handle password modifications and provide the service } >
\end{aligned}$$

We marked the information on C 's credit card with region $\{l_C, l_S\}$ with the intention that only processes at the locations of C and S will be enabled to capture C 's request. Thus, no attacks mounted from other nodes aimed at cancelling the request can take place. Similar considerations do hold for the restricted name pwd that S sends back to C (it represents a secret password shared between processes at C 's and S 's locations).

To make our theoretical framework properly working, we need to control the processes arriving at C 's and S 's locations; this is why our typing discipline requires also nodes to be annotated with regions. Server S can then accept only processes coming from trusted nodes, but it should accept data coming from any user; this is necessary to model a setting where S accepts any service request, while it supplies the service only to accredited users. Thus, $l_S \in r_d^C$ and $l_C \in r_d^S$ (usually, $r_d^S = \top$), while $r_p^C = r_p^S = \emptyset$.

It has to be said that we are implicitly assuming the ability of determining the origin (the source node) of data and processes. By relying on it, we can then check compliance with region annotations.

4.2 Static Inference and Checking

The language presented in the previous section is a mean to program applications where, during the computation, a datum can only appear in localities contained in its region annotation. The runtime semantics can enforce this requirement by performing appropriate checks. These (runtime) checks are necessary because the pattern matching based communication does not permit making any static assumption on the actual structure of tuples hosted by a tuple space (see the discussion on this topic in the previous chapter). To make the semantics as efficient as possible, a preliminary typing phase is introduced. The typing of μKLAIM nets aims at guaranteeing that:

1. a datum $[l]_r$ can be seen at (i.e. can cross) u if $u \in r$
2. a process retrieving a datum $[l]_r$ cannot exhibit l outside r .

$\frac{}{\mathbf{0} > \mathbf{0}}$ <p>(Tc-EMPTY)</p>	$\frac{N_1 > N'_1 \quad N_2 > N'_2}{N_1 \parallel N_2 > N'_1 \parallel N'_2}$ <p>(Tc-NET)</p>
$\frac{N > N'}{(\nu l)N > (\nu l)N'}$ <p>(Tc-RES)</p>	$\frac{r_d, r_p \in \{\top\} \cup 2^{\mathcal{L}} \quad \emptyset \vdash P >_l \emptyset \vdash P'}{l_{r_d::r_p} P > l_{r_d::r_p} P'}$ <p>(Tc-NODE)</p>

Table 4.1: Static Typing for Confining Data and Processes (Nets)

The typing phase performs check 1. statically and annotates bound names occurring in templates with regions to enable efficient execution of check 2. at runtime. To better distinguish the annotations put by the programmers/administrators from those put by the type system, we shall write the latter ones as superscripts and the former ones as subscripts. Hence, the syntax of templates from Table 2.4 becomes

$$T ::= u \mid [!x]^r \mid T_1, T_2$$

Intuitively, $[!x]^r$ states that the datum replacing x will cross at most the localities in r .

The typing procedure for μ KLAIM nets is given in Table 4.1. Net typings are written $N > N'$. The typing step includes a *type checking phase*, to verify that nets are written according to the region annotations therein, and a *type inference phase*, to annotate parameters occurring in templates. Intuitively, the inference phase takes a net N and returns a net N' obtained from N by annotating all the parameters with a region containing the nodes that the received values will cross. E.g., in process $\mathbf{in}(!x)@l.\mathbf{out}([x]_r)@l'$ the declaration $!x$ of variable x must be associated to region r that, in turn, must contain l' (and l , of course). The type checker verifies that each process located at a node l contains only data that can be seen by l (this is done by the judgement $>_l$) and verifies that actions **out** and **eval** send data/code to nodes where the data/code can appear without violating the region annotations.

Judgement $>$ relies on an auxiliary procedure $\Gamma \vdash P >_u \Gamma' \vdash P'$ defined in Table 4.2, where the *type environment* Γ is a finite map from variables to regions. The procedure $\Gamma \vdash P >_u \Gamma' \vdash P'$ determines, for each parameter in P , a region annotation describing the use of that parameter in the continuation process (i.e. where it will be sent); P' is then obtained by decorating P with those annotations. Such regions are determined by the type inference by considering the locality where the process runs (the u decorating $>_u$) and by examining the localities where the variables can appear upon execution of actions **out** and/or **eval**.

Notice, however, that care is needed to avoid inconsistencies on names occurring in a process. As an example, consider the nodes (both of them are legal)

$$\begin{aligned} l &:: \mathbf{in}(!x)@l'.\mathbf{in}(!y)@l''.\mathbf{out}([x]_{\{l,y\}})@l && (\star) \\ l &:: \mathbf{in}(!y)@l'.\mathbf{out}([y]_{\{l,y\}})@l && (\star\star) \end{aligned}$$

<p>(TC-NIL)</p> $\frac{}{\Gamma \vdash \mathbf{nil} \succ_u \Gamma \vdash \mathbf{nil}}$	<p>(TC-PAR)</p> $\frac{\Gamma \vdash C_1 \succ_u \Gamma' \vdash C'_1 \quad \Gamma' \vdash C_2 \succ_u \Gamma'' \vdash C'_2}{\Gamma \vdash C_1 C_2 \succ_u \Gamma'' \vdash C'_1 C'_2}$
<p>(TC-DATUM)</p> $\frac{u \in \text{reg}(t)}{\Gamma \vdash \langle t \rangle \succ_u \Gamma \vdash \langle t \rangle}$	<p>(TC-REC)</p> $\frac{\Gamma \vdash P \succ_u \Gamma' \vdash P'}{\Gamma \vdash \mathbf{rec} X.P \succ_u \Gamma' \vdash \mathbf{rec} X.P'}$
<p>(TC-VAR)</p> $\frac{}{\Gamma \vdash X \succ_u \Gamma \vdash X}$	<p>(TC-NEW)</p> $\frac{\Gamma \vdash P \succ_u \Gamma' \vdash P'}{\Gamma \vdash \mathbf{new}(l).P \succ_u \Gamma' \nearrow^{[l]} \vdash \mathbf{new}(l).P'}$
<p>(TC-IN)</p> $\frac{\Gamma \uplus \{x : \{u\}\}_{x \in \text{bn}(T)} \vdash P \succ_u \Gamma' \uplus \{x : r_x\}_{x \in \text{bn}(T)} \vdash P' \quad (*)}{\Gamma \vdash \mathbf{in}(T)@u'.P \succ_u \Gamma' \nearrow^{\text{bn}(T)} \vdash \mathbf{in}(T')@u'.P'}$	
<p>(TC-READ)</p> $\frac{\Gamma \uplus \{x : \{u\}\}_{x \in \text{bn}(T)} \vdash P \succ_u \Gamma' \uplus \{x : r_x\}_{x \in \text{bn}(T)} \vdash P' \quad (*)}{\Gamma \vdash \mathbf{read}(T)@u'.P \succ_u \Gamma' \nearrow^{\text{bn}(T)} \vdash \mathbf{read}(T')@u'.P'}$	
<p>(TC-OUT)</p> $\frac{\{u, u'\} \subseteq \text{reg}(t) = r \quad \Gamma \vdash P \succ_u \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}(t)@u'.P \succ_u \Gamma' + \{x : r\}_{x \in \text{fn}(t)} \vdash \mathbf{out}(t)@u'.P'}$	
<p>(TC-EVAL)</p> $\frac{u \in \text{reg}(P_1) \quad \Gamma \vdash P_1 \succ_{u'} \Gamma' \vdash P'_1 \quad \Gamma' \vdash P_2 \succ_u \Gamma'' \vdash P'_2}{\Gamma \vdash \mathbf{eval}(P_1)@u'.P_2 \succ_u \Gamma'' + \{x : \{u'\}\}_{x \in \text{fn}(P_1)} \vdash \mathbf{eval}(P'_1)@u'.P'_2}$	
<p>(*) is: T' obtained from T by replacing each $!x$ with $[!x]^{r'_x}$,</p> $\text{for } r'_x = \begin{cases} \top & \text{if } (r_x - \{x\}) \cap \text{bn}(T) = \emptyset \\ r_x - \{x\} & \text{otherwise} \end{cases}$	

Table 4.2: Static Typing for Confining Data and Processes (Components)

Blindly annotating these nodes would result in

$$l :: \mathbf{in}([!x]^{[l,y]})@l'.\mathbf{in}([!y]^{[l]})@l''.\mathbf{out}([x]_{[l,y]})@l$$

$$l :: \mathbf{in}([!y]^{[l,y]})@l'.\mathbf{out}([y]_{[l,y]})@l$$

Here, the occurrence of y in the regions of $!x$ and $!y$ respectively escaped its binder; thus, the y occurring in the annotations are not the same as the y bound in action \mathbf{in} . The solution we designed to accept (\star) is to assign $!x$ the region annotation \top . This is reasonable since $\mathbf{in}([!x]^{[l,y]})@l'$ means ‘retrieve a datum from l' and share it

with a *generic* locality of the net' (indeed y can be dynamically replaced with any locality name). The solution we designed to accept ($\star\star$) is to remove y from $!y$ region annotation and assume that a locality can always occur in the node having that locality as address.

The anomaly (\star) can also appear when using action **new**, e.g. in

$$l :: \mathbf{in}(!x)@l'.\mathbf{new}(l'').\mathbf{out}([x]_{\{l,l''\}})@l$$

This would result in the annotated process

$$l :: \mathbf{in}([!x]_{\{l,l''\}})@l'.\mathbf{new}(l'').\mathbf{out}([x]_{\{l,l''\}})@l$$

where, again, the l'' occurring in the annotation associated to $!x$ by the inference system escapes from the binder **new** that declares l'' . For the sake of simplicity, we overcome this problem like before, i.e. by assigning \top to the region annotation of $!x$.

To rule out anomalies like (\star), in Table 4.2 we use function $\Gamma \nearrow^S$, for S a finite set of names, that is inductively defined as

$$\begin{aligned} \emptyset \nearrow^S &\triangleq \emptyset \\ (\Gamma \uplus \{x : r\}) \nearrow^S &\triangleq \begin{cases} \Gamma \nearrow^S \uplus \{x : \top\} & \text{if } S \cap r \neq \emptyset \\ \Gamma \nearrow^S \uplus \{x : r\} & \text{otherwise} \end{cases} \end{aligned}$$

where \uplus denotes union between environments with disjoint domains.

Function $+$ extends the information of an environment through another environment; formally

$$\begin{aligned} \Gamma + \emptyset &\triangleq \Gamma \\ \Gamma + \{x : r\} &\triangleq \begin{cases} \Gamma' \uplus \{x : r \cup r'\} & \text{if } \Gamma = \Gamma' \uplus \{x : r'\} \\ \Gamma & \text{otherwise} \end{cases} \\ \Gamma + (\{x : r\} \uplus \Gamma') &\triangleq (\Gamma + \{x : r\}) + \Gamma' \end{aligned}$$

Notice that only the entries in the domain of Γ are considered for the extension.

Finally, we exploit the auxiliary function $\mathit{reg}(_)$ that returns the intersection of the data regions occurring in its argument. Its formal definition is

$$\begin{aligned} \mathit{reg}([u]_r) &\triangleq r \\ \mathit{reg}(t_1, t_2) &\triangleq \mathit{reg}(t_1) \cap \mathit{reg}(t_2) \\ \mathit{reg}(\mathbf{nil}) = \mathit{reg}(X) &\triangleq \top \\ \mathit{reg}(\mathbf{new}(l).P) &\triangleq \mathit{reg}(P) - \{l\} \\ \mathit{reg}(\mathbf{out}(t)@u.P) &\triangleq \mathit{reg}(t) \cap \mathit{reg}(P) \\ \mathit{reg}(\mathbf{in}(T)@u.P) = \mathit{reg}(\mathbf{read}(T)@u.P) & \\ &= \mathit{reg}(\mathbf{rec } X.P) \triangleq \mathit{reg}(P) \\ \mathit{reg}(P_1|P_2) = \mathit{reg}(\mathbf{eval}(P_1)@u.P_2) &\triangleq \mathit{reg}(P_1) \cap \mathit{reg}(P_2) \end{aligned}$$

$match(l, [l]_r) = \epsilon$	$\frac{r \subseteq r'}{match([!x]^r, [l]_{r'}) = [l/x]}$
$match(T_1, t_1) = \sigma_1$	$match(T_2, t_2) = \sigma_2$
$\frac{}{match((T_1, T_2), (t_1, t_2)) = \sigma_1 \circ \sigma_2}$	

Table 4.3: Pattern-matching for Confining Data and Processes

Before concluding this section, we briefly comment on some typing rules. It can be easily seen that typing $P_1|P_2$ and $P_2|P_1$ yields the same typing; this relies on commutativity of sets union, since Γ grows up by union of regions. In rule (Tc-NEW), the resulting environment is $\Gamma' \nearrow^{[l]}$ to rule out anomalies like (\star). In rules (Tc-IN) and (Tc-READ), the procedure should type P in the environment Γ extended by associating each $x \in bn(T)$ to region $\{u\}$. At the end of this typing phase, the region annotation r_x calculated for x is associated to the parameter $!x$. Notice that x can occur in x 's region r_x , generating anomalies like ($\star\star$); to avoid this, the annotation for x must be obtained from $r_x - \{x\}$. Moreover, it is possible that x occurs in region annotations for other names bound in the template or within Γ' , because of anomalies like (\star); thus, the environment resulting from this phase must be $\Gamma' \nearrow^{bn(T)}$ and the annotation for $!x$ must be $\{x : (r_x - \{x\})\} \nearrow^{bn(T)}$. In rule (Tc-OUT), the type checker verifies that t can stay both in the hosting locality u and in the target locality u' . The continuation process P is typed in the environment Γ , thus obtaining the annotated process P' and the environment Γ' . Hence, after the typing, the environment must be Γ' extended with the information that the names occurring in t could be seen at $reg(t)$. Similar observations also hold for rule (Tc-EVAL) too; in particular, the check that the process can cross the locality where it is hosted is performed whenever the process is going to migrate.

We deem *well-typed* those nets that successfully passed a typing phase.

Definition 4.2.1 *A net N is well-typed if there exists a net N' such that $N' > N$.*

4.3 Typed Operational Semantics

Like in the previous chapter, we shall only consider for execution *well-formed nets*, i.e. nets where no node is cloned. Moreover, we need some run-time check to confine data and process. Mainly, this is obtained by properly adapting the pattern-matching function (see Table 4.3) and rules (RED-OUT) and (RED-EVAL). The key feature of the pattern-matching is to ensure that, when a name l associated to region r' has to be retrieved by means of a bound variable $!x$ that will be used in r , it must be that $r \subseteq r'$. This fact, together with the static inference, ensures that data are properly used through the computation.

The structural congruence relation, \equiv , is modified to include node annotations and to remove rule (STR-CLONE), like in the previous chapter. The reduction relation

(RED-OUT)	$l \in r'_d$
$l_{r_d::r_p} \mathbf{out}(t)@l'.P \parallel l'_{r'_d::r'_p} C' \mapsto l_{r_d::r_p} P \parallel l'_{r'_d::r'_p} C' \mid \langle t \rangle$	
(RED-EVAL)	$l \in r'_p$
$l_{r_d::r_p} \mathbf{eval}(Q)@l'.P \parallel l'_{r'_d::r'_p} C' \mapsto l_{r_d::r_p} P \parallel l'_{r'_d::r'_p} C' \mid Q$	
(RED-IN)	$match(T, t) = \sigma$
$l_{r_d::r_p} \mathbf{in}(T)@l'.P \parallel l'_{r'_d::r'_p} \langle t \rangle \mapsto l_{r_d::r_p} P\sigma \parallel l'_{r'_d::r'_p} \mathbf{nil}$	
(RED-READ)	$match(T, t) = \sigma$
$l_{r_d::r_p} \mathbf{read}(T)@l'.P \parallel l'_{r'_d::r'_p} \langle t \rangle \mapsto l_{r_d::r_p} P\sigma \parallel l'_{r'_d::r'_p} \langle t \rangle$	
(RED-NEW)	
$l_{r_d::r_p} \mathbf{new}(l').P \mapsto (\nu l')(l_{r_d \cup \{l'\}::r_p \cup \{l'\}} P \parallel l'_{r_d \cup \{l'\}::r_p \cup \{l'\}} \mathbf{nil})$	
(RED-SPLIT)	
$l_{r_d::r_p} C_1 \parallel l_{r_d::r_p} C_2 \parallel N \mapsto (\nu \tilde{l})(l_{r'_d::r'_p} C_1 \parallel l_{r_d::r_p} C_2 \parallel N')$	
$l_{r_d::r_p} C_1 \mid C_2 \parallel N \mapsto (\nu \tilde{l})(l_{r'_d::r'_p} C_1 \mid C_2 \parallel N')$	
with rules (RED-PAR), (RED-RES) and (RED-STRUCT) from Table 2.3	

Table 4.4: Operational Semantics for Confining Data and Processes

adapts the rules of Table 2.5 as shown in Table 4.4; we only remark a few points. In rules (RED-OUT) and (RED-EVAL) a datum/process can be put at the target of the **out/eval** only if such a node accepts the datum/process (i.e. $l \in r'_d$ and $l \in r'_p$). This is necessary to prevent an untrusted node l to send data/code to l' . Again, notice that no static check could enforce this property without loss of expressivity. Rules (RED-IN) and (RED-READ) behave like before; just notice that the new definition of the pattern-matching must be used and that the application of a substitution to a process P also acts on the region annotations therein.

Like in the previous chapter, we can easily prove that the introduction of rule (RED-SPLIT) and the removal of rule (STR-CLONE) are enough to ensure that net well-formedness is preserved along reductions (see Proposition 3.2.3 whose proof can be easily adapted to the present setting).

To conclude, let us come back to the example presented in Section 4.1. By reasonably assuming that the password modification is carried on by only involving l_C and l_S , the inference system annotates P_C as follows:

$$P'_C \triangleq \mathbf{out}(l_C, [cc_info]_{\{l_C, l_S\}})@l_S.\mathbf{in}([!y]^{l_C, l_S})@l_C.\dots$$

Similarly, if we assume that credit card checking is performed locally by the server and never used anymore, P_S is annotated as:

$$P'_S \triangleq \mathbf{rec} X.\mathbf{in}([!x_1]^{l_S}, [!x_2]^{l_S})@l_S.\dots.\mathbf{new}(p).\mathbf{out}([p]_{\{x_1, l_S\}})@x_1.\dots$$

Now, the dynamic checks of rule (RED-IN) are respected; thus, the resulting net can evolve as expected:

$$\begin{aligned} & l_C r_d^C :: r_p^C P'_C \parallel l_S r_d^S :: r_p^S P'_S \\ \mapsto^* & l_C r_d^C :: r_p^C \mathbf{in}([!y]^{l_C, l_S})@l_C.\dots \parallel \\ & l_S r_d^S :: r_p^S P'_S \mid \langle \mathit{check\ cc_info} \rangle . \mathbf{new}(p).\mathbf{out}([p]_{\{l_C, l_S\}})@l_C.\dots \\ \mapsto^* & l_C r_d^C :: r_p^C \langle \mathit{modify\ password\ p\ and\ access\ the\ service} \rangle \parallel \\ & l_S r_d^S :: r_p^S P'_S \mid \langle \mathit{handle\ pwd\ modifications\ and} \\ & \quad \mathit{provide\ the\ service} \rangle \end{aligned}$$

4.4 Type Soundness

Our main results state that well-typedness is preserved along reductions and that well-typed nets do respect region annotations. The proof of the first result, i.e. *subject reduction*, is similar to the corresponding proofs in the previous chapter; thus, we only highlight the differences. About the second result, i.e. *safety*, we need a more sophisticated theory that we shall discuss with some details later on.

Lemma 4.4.1 (Subject Congruence) *If N is well-typed and $N \equiv N'$ then N' is well-typed.*

Lemma 4.4.2 (Substitutivity) *If $\Gamma \uplus \{x_i : r_i\}_{i \in I} \vdash P \succ_u \Gamma' \uplus \{x_i : r'_i\}_{i \in I} \vdash P'$ and $\sigma = [^l_i/x_i]_{i \in I}$, then $\Gamma\sigma \vdash P\sigma \succ_{u\sigma} \Gamma'\sigma \vdash P'\sigma$.*

Proof: Notice that it suffices to prove the claim for $|I| = 1$; indeed, a straightforward induction on $|I|$, together with the fact that $[^l_i/x_i]_{i=1,\dots,n} = [^l_1/x_1] \circ \dots \circ [^l_n/x_n]$, can cover the more general case.

Thus, we consider only $\Gamma \uplus \{x : r\} \vdash P \succ_u \Gamma' \uplus \{x : r'\} \vdash P'$ and we proceed by induction on the length of the inference used to derive it. The base case is when rules (TC-NIL) or (TC-DATUM) are used: in both cases it is trivial to conclude. Let us consider the inductive case and reason by case analysis on the last rule used to infer the judgement. We explicitly show the most significant case, i.e. when using (TC-IN); the remaining cases are similar or easier. By definition, $P = \mathbf{in}(T)@u'.Q$ and $P' = \mathbf{in}(T')@u'.Q'$, where $\Gamma \uplus \{x : r\} \uplus \{y : \{u\}\}_{y \in \mathit{bn}(T)} \vdash Q \succ_u \Gamma'' \uplus \{x : r''\} \uplus \{y : r''\}_{y \in \mathit{bn}(T)} \vdash Q'$ and T' is defined by (*) in Table 4.2. By hypothesis, $x \notin \mathit{bn}(T)$; thus, by induction,

$$\Gamma\sigma \uplus \{y : \{u\}\}_{y \in \mathit{bn}(T)} \vdash Q\sigma \succ_{u\sigma} \Gamma''\sigma \uplus \{y : r''\}_{y \in \mathit{bn}(T)} \vdash Q'\sigma$$

Moreover, $\Gamma' = \Gamma'' \nearrow^{bn(T)}$ and thus $\Gamma'\sigma = (\Gamma'' \nearrow^{bn(T)})\sigma = (\Gamma''\sigma) \nearrow^{bn(T)}$. Hence, by using rule (Tc-IN), we can conclude. ■

Theorem 4.4.3 (Subject Reduction) *If N is well-typed and $N \mapsto N'$, then N' is well-typed.*

Proof: The proof proceeds by induction on the length of the inference. We only consider the most peculiar cases.

(RED-OUT). By hypothesis, $N = l_{r_d::r_p} \mathbf{out}(t)@l'.P \parallel l'_{r'_d::r'_p} C'$ and there exists a net M such that $M > N$. By definition, $M = l_{r_d::r_p} \mathbf{out}(t)@l'.Q \parallel l'_{r'_d::r'_p} C''$ where $\emptyset \vdash \mathbf{out}(t)@l'.Q >_l \emptyset \vdash \mathbf{out}(t)@l'.P$ and $\emptyset \vdash C'' >_{l'} \emptyset \vdash C'$. By the premises of rule (Tc-OUT), $\emptyset \vdash Q >_l \emptyset \vdash P$ and $l' \in r$. This suffices to conclude that $N' = l_{r_d::r_p} P \parallel l'_{r'_d::r'_p} P' | \langle t \rangle$ is well-typed.

(RED-IN). By hypothesis, N results from the typing of a net $M = l_{r_d::r_p} \mathbf{in}(T)@l'.Q \parallel l'_{r'_d::r'_p} \langle t \rangle$. By assuming that $match(T, t) = \sigma$, that must hold because of the premise of rule (RED-IN), the main thing to prove is that the well-typedness of $l_{r_d::r_p} \mathbf{in}(T)@l'.P$ implies the well-typedness of $l_{r_d::r_p} P\sigma$. By the premise of rule (Tc-IN), it holds that

$$\{x : \{l\}_{x \in bn(T)}\} \vdash Q >_l \{x : r_x\}_{x \in bn(T)} \vdash P$$

Hence, by Lemma 4.4.2, $\emptyset \vdash Q\sigma >_l \emptyset \vdash P\sigma$; this suffices to conclude.

(RED-SPLIT). By hypothesis, we have that $N = l_{r_d::r_p} C_1|C_2 \parallel N''$ results from the typing of a net $M = l_{r_d::r_p} D_1|D_2 \parallel M''$. In particular, $\emptyset \vdash D_1|D_2 >_l \emptyset \vdash C_1|C_2$ that, by rule (Tc-PAR), implies that $\emptyset \vdash D_1 >_l \Gamma \vdash C_1$ and $\Gamma \vdash D_2 >_l \emptyset \vdash C_2$. However, Γ must be \emptyset as well; indeed, it can be easily checked that $\Gamma_1 \vdash C >_l \Gamma_2 \vdash C'$ implies $dom(\Gamma_1) = dom(\Gamma_2)$. Hence, $l_{r_d::r_p} C_1 \parallel l_{r_d::r_p} C_2 \parallel N''$ is well-typed and, by induction, $(\widetilde{\nu}l)(l_{r'_d::r'_p} C'_1 \parallel l_{r_d::r_p} C'_2 \parallel N')$ is well-typed. This implies that $(\widetilde{\nu}l)(l_{r'_d::r'_p} C'_1|C'_2 \parallel N')$ is well-typed, as required. ■

We now turn to type safety. As we have already said, it states that well-typedness guarantees absence of immediate violations of data regions. However, the wanted safety property requires that data regions are respected along all possible computations. To properly formalise this property we need to define a finer semantics. Indeed, deeming a net to be safe when “for any node $l_{r_d::r_p} P$ it holds that l occurs in the region of each datum in P ” would not be satisfactory because the regions annotating data disappear upon data withdrawal. Thus, it would become impossible to formalise the requirement that the region specification associated to a datum when it is produced is respected during all the datum life-time (i.e. also after its retrieval). For example, consider the net

$$N = l_{r_d::r_p} \mathbf{in}(!x]^{r'})@l'.P \parallel l'_{r'_d::r'_p} \langle [l'']_r \rangle$$

Upon execution of action **in**, the net becomes $N' = l_{r_d :: r_p} P' \parallel l'_{r'_d :: r'_p} \mathbf{nil}$, where $P' = P[l''/x]$. Now, the new occurrences of l'' in P' are *not* annotated anymore with region r . Hence, in N' we have no mean to formalise the statement that l can use l'' by respecting the original annotation r .

To overcome this problem, we design a *tagged language*, where each occurrence of a locality in a process is tagged with a region determining its visibility. To this aim, we slightly adapt the syntax of μKLAIM , by letting

$$u ::= [l]_r \mid x$$

We can now formalise when a net is safe. To this aim, we extend function reg defined in Section 4.2 by taking into account also the locality tags when calculating the region intersection. For example, $reg(\mathbf{out}([l]_{r_1}]_{r_2})@[l']_{r_3}.P) = r_1 \cap r_2 \cap r_3 \cap reg(P)$.

Definition 4.4.4 (Safety) *A net N is safe if, for any $l_{r_d :: r_p} C$ in N , it holds that $l \in reg(C)$.*

The tagged operational semantics generalises that in Table 4.4; indeed, processes like $\mathbf{out}([l]_{r_1}]_{r_2})@[l']_{r_3}$ or $\mathbf{in}([l]_{r_1})@[l']_{r_2}$ can evolve. These terms may arise upon application of substitutions that now map names into names tagged with regions. We let the application of the substitution to a region to replace names with plain names only (hence omitting their tags) thus ensuring that regions are still sets of names. The reduction relation, however, ignores the tags and considers tagged names as plain ones. This should have been somehow expected because, as we said before, the only role of tags is to enable formalising and checking that a net is safe. To avoid confusion, we use the arrow \mapsto to relate tagged terms.

The typing procedure for tagged terms is denoted by \gg and its most significant rules are given in Table 4.5 (the other ones are smooth adaptations of those in Table 4.2). We use functions $pid(u)$ and $reg(u)$ to denote, respectively, the plain identifier and the region of the tagged identifier u . The intuition underlying \gg is that, whenever an identifier occurs at a locality, the locality must be included in the region tagging the identifier.

Given a plain net N , we use $tag(N)$ to denote the set containing all the well-typed (w.r.t. \gg) tagged nets obtained by tagging localities in N . Given a tagged net N , we denote with $untag(N)$ the plain net obtained from N by removing all the locality tags. Notice that $tag(N)$ is not empty because it contains at least the net obtained by tagging each locality in N with \top . We call the latter net the *outset tagging* of N .

Predictably, the tagged language and the original one are strongly related. Moreover, the typing of tagged terms is preserved along (tagged) reductions. The following results formalise these properties.

$\frac{pid(u) \in reg(u') \quad \{pid(u), pid(u')\} \subseteq reg(t) = r \quad \Gamma \vdash P \gg_u \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}(t)@u'.P \gg_u \Gamma' + \{x : r\}_{x \in fn(t)} \vdash \mathbf{out}(t)@u'.P'}$
$\frac{pid(u) \in reg(u') \cap reg(P_1) \quad \Gamma \vdash P_1 \gg_{u'} \Gamma' \vdash P'_1 \quad \Gamma' \vdash P_2 \gg_u \Gamma'' \vdash P'_2}{\Gamma \vdash \mathbf{eval}(P_1)@u'.P_2 \gg_u \Gamma'' + \{x : \{pid(u')\}\}_{x \in fn(P_1)} \vdash \mathbf{eval}(P'_1)@u'.P'_2}$
$\frac{pid(u) \in reg(u') \quad \Gamma \uplus \{x : \{pid(u)\}\}_{x \in bn(T)} \vdash P \gg_u \Gamma' \uplus \{x : r_x\}_{x \in bn(T)} \vdash P' \quad (*)}{\Gamma \vdash \mathbf{in}(T)@u'.P \gg_u \Gamma' \nearrow^{bn(T)} \vdash \mathbf{in}(T')@u'.P'}$
$\frac{pid(u) \in reg(u') \quad \Gamma \uplus \{x : \{pid(u)\}\}_{x \in bn(T)} \vdash P \gg_u \Gamma' \uplus \{x : r_x\}_{x \in bn(T)} \vdash P' \quad (*)}{\Gamma \vdash \mathbf{read}(T)@u'.P \gg_u \Gamma' \nearrow^{bn(T)} \vdash \mathbf{read}(T')@u'.P'}$
<p>(*) is defined like in Table 4.2</p>

Table 4.5: Tagged Typing Rules

Proposition 4.4.5 1. If $N \gg M$ then $untag(N) > untag(M)$.

2. If $N > M$, then for all $M' \in tag(M)$ there exists $N' \in tag(N)$ such that $N' \gg M'$.

3. If $N \mapsto N'$ then $untag(N) \mapsto untag(N')$.

Proof: All properties easily follow from definitions of \gg and \mapsto . ■

Corollary 4.4.6 (Tagged Subject Reduction) If N is a well-typed tagged net and $N \mapsto N'$, then N' is a well-typed (tagged) net.

Proof: By Propositions 4.4.5(1) and (3), it holds that $untag(N)$ is well-typed and that $untag(N) \mapsto untag(N')$. Because of Theorem 4.4.3, this implies that $untag(N')$ is well-typed and, by Proposition 4.4.5(2), we can conclude. ■

We are now ready to prove the type safety theorem.

Theorem 4.4.7 (Type Safety) If N is a well-typed tagged net then N is safe.

Proof: By definition, N is a well-typed (tagged) net if there exists a net M such that $M \gg N$. The proof proceeds by induction on the length of the inference leading to this judgement and heavily relies on checking the premise $pid(u) \in reg(u')$ contained in each rule of Table 4.5. ■

Corollary 4.4.8 (Type Soundness) *Let N be a (plain) well-typed net and N' be its outset tagging. Then $N' \mapsto^* N''$ implies that N'' is safe.*

Proof: By Proposition 4.4.5(2) and by the fact that $N' \in \text{tag}(N)$, it holds that N' is a well-typed tagged net. We now proceed by induction on the length of \mapsto^* . The base case is Theorem 4.4.7; the inductive case trivially follows by exploiting Corollary 4.4.6. ■

The results given above can be generalised by requiring only a subnet of the whole net to be well-typed. By using the convention that absence of a region annotation means \top , a not well-typed net can be executed according to the (tagged versions of) rules in Table 4.4 by safely considering all its variable annotations as \top . We call r -subnet of N the net formed by all the nodes $l_{r_d}::r_p P$ in N such that $\{l\} \cup r_d \cup r_p \subseteq r$. Notice that such a net is not necessarily defined for all r ; of course it is always defined for $r = \top$ and coincides with N (in this case Theorem 4.4.9 coincides with Corollary 4.4.8).

Theorem 4.4.9 (Localised Type Soundness) *Let N be a plain net and N' be its outset tagging. If the r -subnet of N' is defined and well-typed, and if $N' \mapsto^* N''$, then the r' -subnet of N'' is defined and safe, where $r' = r \cup L$ and L is the set of node addresses created during the computation.*

Proof: By exploiting Theorem 4.4.7, we only need to show that the r' -subnet of N'' is defined and well-typed. We just consider the case for $N' \mapsto N''$; the more general case is recovered by using an inductive argument similar to that in Corollary 4.4.7. The proof proceeds like that of Theorem 4.4.3. Just notice that, when the operational rule used to infer the reduction is (the tagged version of) (RED-OUT) or (RED-EVAL) resp., the premise $l \in r'_d$ or $l \in r'_p$ respectively turns out to be crucial to maintain well-typedness. Moreover, the only non trivial case for establishing if the r' -subnet is defined is when the operational rule used is (RED-NEW). In this case, the claim is easily proved since the new node is assigned the regions of the creating one. ■

To conclude, we want to remark that the language can be easily extended to enable explicit specification of the regions of the new nodes. In this case, existence of the r' -subnet should be ensured by adding a premise to rule (RED-NEW) requiring that the regions of the new node are included in those of the creating node.

4.5 Example: Implementing a Multiuser System

In this section we use the framework presented so far to program a simple but meaningful example. We present the behaviour of a simple UNIX-like multiuser system, where users can login (exploiting a password-based approach) and use the system functionalities, which consist in reading/writing files or executing programs. For the sake of presentation, we shall present the system in three steps and,

finally, we shall merge them together. Let l_S be the address of the server, \top be its data trust region and \emptyset be its process trust region (thus no user can spawn code to l_S).

User Identification. We start with programming the identification of different users via passwords. Localities play the role of user IDs. Let l_p be a private repository used by l_S to record the registered users and their passwords. Thus, l_p hosts the tuples

$$\langle l_1, [pwd_1]_{\{l_1, l_p, l_S\}} \rangle \mid \dots \mid \langle l_n, [pwd_n]_{\{l_n, l_p, l_S\}} \rangle$$

Let l be a user wanting to log in l_S . If l is already known to l_S (i.e. it is one of the l_i s), then l can use a process like

$$\mathbf{out}(\text{"login"}, l, [pwd]_{\{l, l_S\}})@l_S. \mathbf{in}(\text{"logged"})@l_S. \dots$$

for communicating with the server process

$$\text{Login}(l_p) \triangleq \mathbf{rec } X. \mathbf{in}(\text{"login"}, !u, !z)@l_S. (X \mid \mathbf{read}(u, z)@l_p. \mathbf{out}([\text{"logged"}]_{\{l_S, u\}})@l_S)$$

Intuitively, l requires a connection by sending its user ID (its locality) and its password; the server checks if this information is correct and sends back an ack, activating the continuation of the computation at l . Notice that the region annotations of pwd and “logged” rule out attacks of a nasty intruder aimed at cancelling the request of login or the corresponding ack, and preserve the secrecy of the password.

If the user is not registered at l_S yet, he can send an “hello” request to the server containing its address and wait for a password

$$\mathbf{out}([\text{"hello"}]_{\{l, l_S\}}, l)@l_S. \mathbf{in}(\text{"registered"}, !pwd)@l_S. \dots$$

The server then handles this request with the process

$$\text{NewUser}(l_p) \triangleq \mathbf{rec } X. \mathbf{in}(\text{"hello"}, !u)@l_S. (X \mid \mathbf{new}(pwd). \mathbf{out}(u, [pwd]_{\{u, l_S, l_p\}})@l_p. \mathbf{out}(\text{"registered"}, [pwd]_{\{l_S, u\}})@l_S)$$

Of course, a locality l' different from l can send to l_S a request for a new password pretending to be l : the only difference with the “hello” message given above is that the message now should contain also l' in the data region. However, the server will report the new password to l and the region associated to the password will ensure that pwd will not leave l . Thus, l' cannot withdraw pwd : it can only try to send a process to l for acting at l with the new password. This can be possible only if l trusts l' , implying that l accepts this ‘suspicious’ activity of l' .

We now show the use of our typing theory in the setting just presented. In particular, we give evidence of how we can prevent attacks aimed at cancelling messages and activities of malicious users pretending to play the role of other users.

- Let l_{canc} be a locality hosting a process that aims at interfering with the login procedure above by performing action $\mathbf{in}(\text{"hello"}, !x)@l_S$. In this way, it removes the *hello* message sent by an unregistered user l willing to be connected with the server l_S . The system is modelled as follows

$$\begin{aligned}
& l_{\text{canc}} :: \mathbf{in}(\text{"hello"}, !x)@l_S.DONE \parallel l_S :: \text{NewUser}(l_p) \\
& \parallel l :: \mathbf{out}([\text{"hello"}]_{\{l, l_S\}}, l)@l_S.\mathbf{in}(\text{"registered"}, !pwd)@l_S. \dots \\
& \quad \mapsto l_{\text{canc}} :: \mathbf{in}(\text{"hello"}, !x)@l_S.DONE \\
& \quad \parallel l :: \mathbf{in}(\text{"registered"}, !pwd)@l_S. \dots \\
& \quad \parallel l_S :: \text{NewUser}(l_p) \mid \langle [\text{"hello"}]_{\{l, l_S\}}, l \rangle \\
& \quad \not\mapsto l_{\text{canc}} :: DONE \parallel l :: \mathbf{in}(\text{"registered"}, !pwd)@l_S. \dots \\
& \quad \parallel l_S :: \text{NewUser}(l_p)
\end{aligned}$$

Notice that the last transition cannot take place. As expected, the intruder running at l_{canc} is not enabled to withdraw the tuple $\langle [\text{"hello"}]_{\{l, l_S\}}, l \rangle$ because $l_{\text{canc}} \notin \{l, l_S\}$ (see the definition of pattern-matching in Table 4.3).

- Let now l_{pret} be a locality pretending to act on behalf of l , by trying to acquire a log to l_S under the identity of l . Let us examine the possible evolutions of the system:

$$\begin{aligned}
& l_{\text{pret}} :: \mathbf{out}(\text{"hello"}, l)@l_S.\mathbf{in}(\text{"registered"}, !pwd)@l_S.DONE \\
& \parallel l_S :: \text{NewUser}(l_p) \\
& \quad \mapsto \mapsto \mapsto \mapsto \mapsto l_{\text{pret}} :: \mathbf{in}(\text{"registered"}, !pwd)@l_S.DONE \\
& \quad \parallel l_S :: \text{NewUser}(l_p) \mid \langle \text{"registered"}, [pwd]_{\{l_S, l\}} \rangle \\
& \quad \not\mapsto l_{\text{pret}} :: DONE \parallel l_S :: \text{NewUser}(l_p)
\end{aligned}$$

Again, the last reduction cannot take place because $l_{\text{pret}} \notin \{l_S, l\}$. The only way for l_{pret} to withdraw the tuple $\langle \text{"registered"}, [pwd]_{\{l_S, l\}} \rangle$ is to spawn a process to l (if it exists in the net) executing action $\mathbf{in}(\text{"registered"}, !pwd)@l_S$ (that would be enabled, because $l \in \{l_S, l\}$). Such a migration, however, should be authorised by l (indeed, it can take place only if $l_{\text{pret}} \in r_p^l$, where r_p^l is the node region controlling migrations to l).

The File System. We now consider a server handling a file system where different users can write/read data. Let l_f be a private repository used by l_S to store the files. A file named N , whose content is the string S , that can be read by users in r and written by users in r' , is stored in l_f as the component

$$C_N \triangleq \langle N, [\text{"read"}]_{r \cup \{l_S, l_f\}}, [\text{"written"}]_{r' \cup \{l_S, l_f\}} \rangle \mid \langle N, S \rangle$$

Intuitively, *read* and *written* are just dummy data used to properly store regions r and r' . Then, the server handles requests for reading and writing files with the

The evolution of user l is

$$\begin{aligned}
l &:: \mathbf{out}(\text{"read"}, l, FILE)@l_S . \mathbf{in}(\text{"read"}, FILE, !cont)@l.P \\
&\parallel l_f :: C_{FILE} \parallel l_S :: TRead(l_f) \\
\mapsto \mapsto & \quad l :: \mathbf{in}(\text{"read"}, FILE, !cont)@l.P \parallel l_f :: C_{FILE} \\
&\quad \parallel l_S :: TRead(l_f) \mid \mathbf{read}(FILE, [!z_r]^{l_f, l_S, l}, [!z_w]^{l_S})@l_f. \dots
\end{aligned}$$

Now, action $\mathbf{read}(FILE, [!z_r]^{l_f, l_S, l}, [!z_w]^{l_S})@l_f$ is enabled, because $\{l_f, l_S, l\}$ is a subset of ρ ; thus, the content of $FILE$ will be transferred to l that, in turn, will be enabled to retrieve it (by binding $content$ to the variable $cont$) and use it in P . Notice that, if a user $l' \notin \rho$ had tried to carry on the same task, these actions would not have been enabled, since $\{l_f, l_S, l'\} \not\subseteq \rho$.

Executing Code-on-Demand. In this last setting, a user can dynamically download some code from the server to perform a given task. The server stores all the downloadable processes as executable (named) files in a private locality l_c . For each executable file named N , whose code is P and that is downloadable by nodes in r , the server stores in l_c the component

$$\begin{aligned}
C_N &\triangleq \langle N, [\text{"downloaded"}]_{r \cup \{l_S, l_c\}} \rangle \quad \mid \\
&\quad \mathbf{rec} X. \mathbf{in}(req, N, !u)@l_c. (X \mid \mathbf{read}(N, !z_e)@l_c. \\
&\quad \mathbf{eval}(\mathbf{eval}(\mathbf{out}([z_e]_{l_c, l_S, u}, N)@u.P)@u)@l_S)
\end{aligned}$$

Then, when a user wants to download some code, the server handles its request with the process

$$Execute(l_c) \triangleq \mathbf{rec} X. \mathbf{in}(\text{"execute"}, !u, !n)@l_S . \mathbf{out}(req, n, u)@l_c.X$$

Notice that l_c cannot directly send P for execution to u because (the locality associated to) u cannot have l_c in its trust region (since l_c is fresh). Thus, P must firstly cross l_S and then, if l_S is in the process trust region of u (which we assume it is the case), the code-on-demand procedure successfully terminates, by also reporting an ack to the user.

The System. Finally, we can put together the activities shown so far to obtain the implementation of the whole server. Thus, the (not yet typed) initial configuration of l_S would be

$$\begin{aligned}
l_S \text{ } \tau :: \emptyset & \quad \mathbf{new}(l_p). \mathbf{new}(l_f). \mathbf{new}(l_c). \\
& \quad \langle \text{set up } l_p \text{ with the identities and passwords of the users} \rangle . \\
& \quad \langle \text{set up } l_f \text{ with the data of the file system} \rangle . \\
& \quad \langle \text{set up } l_c \text{ with the downloadable processes} \rangle . \\
& \quad (\mathbf{NewUser}(l_p) \quad \mid \quad \mathbf{Login}(l_p) \quad \mid \quad \mathbf{Read}(l_f) \quad \mid \\
& \quad \mathbf{Write}(l_f) \quad \mid \quad \mathbf{Execute}(l_c))
\end{aligned}$$

Our example simplifies UNIX behaviour in two major aspects: first, we did not require that a user must login before using the functionalities offered by the system;

second, the files/programs are put by the system and not by the users. Both these choices were driven by the aim of simplifying the presentation, but our setting could be easily enriched with more refined and realistic features.

Finally, we want to remark that, by exploiting the dummy data “*downloaded*”, “*read*” and “*written*”, we have been able to enforce an access control policy by only using region annotations. This confirms that, in spite of its simplicity, the approach we have presented in this chapter is very powerful.

4.6 Discussion and Related Work

In this chapter, we presented a programming notation aiming at protecting the secrecy of both host and process data in global computing applications. By exploiting a preliminary compilation and some (unavoidable) runtime checks, we ensured that data can cross only nodes which are allowed to see them. The technique developed is good enough to assume only a local knowledge of the net during the compilation and work even when misbehaving entities are present in the system.

We want to remark that our theory permits to naturally implement a security mechanism based on *sandboxes*. We already noticed that nodes can be seen as logical partitions of a single physical machine. By exploiting this intuition, one can split each machine into an appropriate number of nodes each with its own security policy. In this way, fine grained security policies can be programmed to guarantee that untrusted processes (e.g., coming from unchecked nodes) are accepted only at dedicated nodes and that from these nodes remote operations and spawning of threads are not permitted.

The system we presented here is quite simple and easily implementable (types are just sets and operations on types are unions, intersections, subset inclusions, ...). Also its runtime semantics is reasonable, since it only involves efficiently implementable operations on sets. Moreover, by accepting more runtime checks, node regions could be handled more dynamically by defining primitives for adding and removing nodes from regions (in this way, e.g., nodes could choose whether to trust newly created ones). The problem is that, in this richer framework, none of the two guarantees illustrated at the beginning of Section 4.2 could be issued after static checks. The type system would then only permit inferring the regions of the arguments of process actions, and render dynamic checks more efficient.

Related Work

Our work has been inspired by that on Confined- λ [98], a higher-order functional language that supports distributed computing by allowing expressions at different localities to communicate via channels. To limit the movement of values, programmers can assign a type (i.e. a region) to them; a type system is defined that guarantees that each value can roam only within the allowed region. There are however some differences with our approach. First of all, we consider tuple spaces, another

communication media that require a more dynamic typing mechanism. Then, we permit annotating only the relevant data while in [98] a programmer must declare a type for any constant, function and channel. When typing a net, we do not rely on any form of global knowledge of the system; only the annotations in the process are considered. On the contrary, the type system in [98] assumes a global typing environment for handling shared channels; this somehow conflicts with the features of a global computing setting. Finally, we also give ‘localised’ formulations of the soundness theorem stating that well-typedness of a given subnet is preserved also in presence of untyped contexts. This is a crucial property for global computing systems where little assumptions on the behaviour of the context can be made.

The group types, originally proposed for the Ambients calculus [38] and then recast to the π -calculus [35], have purposes similar to our region annotations. A group type is just a name that can be dynamically created but cannot be communicated (i.e. the scope of a group name cannot be extended). It permits to control name visibility in different regions of a net: a fresh name belonging to a fresh group can never be communicated to any process outside the scope of the group. Group types can then be used to handle processes and ambients movements, and in general to prevent accidental or malicious leakage of private names without using more complex dependent types. However, differently from our approach, when exploiting group types some global knowledge is still necessary for taking into account the types of the names occurring free in a net.

Group types have also been used in region-based memory management where the focus is on efficiency, rather than on distribution and mobility. For instance, in [55] a connection between memory regions and group types is established and a variant of the π -calculus equipped with group types is used as a device to simplify the proof of correctness of dynamic memory management.

Finally, we want to consider a lower level approach to protect visibility of data via *encryption*. Encrypted data can appear everywhere in the net, but can be effectively used only by those users that know the decryption key. At an abstract level, we can consider the content of an encrypted message to be visible only within the region containing the nodes knowing the decryption key. Thus, it might appear that our approach could be implemented by resorting to cryptographic primitives. However, we would like to stress an important difference. When encryption is used, the producer of encrypted data can control the access to (plain) data only by controlling visibility of the decryption key. But this can be hardly controlled: once a decryption key has been passed on, information leakage can reveal the key, thus breaking the controllability of data. By exploiting our approach, the data producer can decide in advance which are the users enabled to access the data; this information is preserved during any evolution of the system. However, it should be noticed that indirect information flows can be generated; for an account of these problems and some possible solutions we refer the interested reader to [84, 89].

Chapter 5

Behavioural Theories

Programming computational infrastructures available globally for offering uniform services has become one of the main issues in computer science, as we have largely argued in the Introduction. On the foundational side, the demand is on the development of tools and techniques to build safer and trustworthy global systems, to analyse their behaviour, and to demonstrate their conformance to given specifications. Indeed, theoretical models and calculi can provide a sound basis for building systems which are “sound by construction” and which behave in a predictable and analysable manner. The crux is to equip the abstractions for global computers with effective tools to support development and certification of global computing applications.

Semantic theories, needed for stating and proving observable properties, should reflect all (or most of) the distinctive features of global systems at user’s application level; hence, they should ignore issues such as routing or network topology, because they are hardly observable at the user level.

In this chapter, we define some sensible semantics theories for μKLAIM and develop for them some tractable proof techniques. Given the direct correspondence of μKLAIM with X-KLAIM , we believe that the tractable behavioural equivalences we develop here provide powerful tools to write sound programs for global computers. First, programs written in X-KLAIM are mapped down to μKLAIM ; here they are verified, by using behavioural equivalences to formalise and prove properties; finally, they can run on an actual global computer, like the Internet, by exploiting their Java-based translation [12].

We develop the semantic theories by defining behavioural equivalences over terms as the maximal congruences induced by some basic *observables* that are dictated by the relevant features of global computers. The approach can be summarised as follows:

1. Define a set of observables (values, normal forms, actual communications, ...) to which a term can evaluate by means of successive reductions.
2. Define a basic equivalence over terms by stating that two terms are equivalent if and only if they exhibit the same set of basic observables.

3. Consider the largest congruence over the language induced by the basic equivalence or by its co-inductive closure.

A similar approach has already been used to study models of concurrent systems (e.g., CCS [115, 15] and π -calculus [133, 5]). Obviously, the designation of the basic observables is critical. Thus, we draw inspiration from everyday experience: a user can observe the behaviour of a global computer (at the application level) by testing

- i.* whether a specific site is up and running (i.e., it provides some data of any kind),
- ii.* whether a specific information is present in (at least) a site, or
- iii.* whether a specific information is present in a specific site.

Other calculi for global computers rely on (barbed) congruences induced by similar observables: for example, Ambient [41] uses a barb that is somehow related to *i.* above, while the barbs in $D\pi$ -calculus [85] are strongly related to *iii.*

A question that naturally arises is whether these observables yield ‘interesting’ congruences. The three basic observables, together with the discriminating power provided by μKLAIM contexts, all yield the same congruence, when used similarly. This is for us already an indication of the robustness of the resulting semantic theories. Moreover, as we will show, the observables are still sufficiently powerful to give rise to interesting semantic theories also when considering lower-level features like, e.g., failures. Due to its intuitive appeal, in the rest of this chapter we shall mainly use the first kind of observable.

A major drawback of the approach relying on basic observables and context closures is that the resulting congruences are defined via universal quantification over all language contexts, and this makes their checking very hard. It is then important to devise proof techniques that avoid such quantification. We shall define a labelled transition system (with labels indicating the performed action) and exploit the labels to avoid quantification over contexts. We shall present tractable characterisations of two ‘touchstone’ congruences, namely *barbed congruence* and *may testing*, in terms of (non-standard) labelled *bisimilarity* and *trace* equivalence, respectively. In doing this, we have to face the problems raised by the presence of explicit localities and by the fact that μKLAIM is asynchronous (both in the communication and in the mobility paradigm) and higher-order (because processes can migrate).

5.1 Touchstone Equivalences

In this section we present (weak) equivalences yielding sensible semantic theories for μKLAIM . The approach we follow relies on the definition of an *observation* (also called *barb*) that intuitively formalises the possible interactions of a process.

We use observables to define equivalence relations that identify those nets that cannot be taken apart by any basic observation after any reduction in any context. Notationally, we shall use \Longrightarrow to denote the reflexive and transitive closure of \mapsto .

Definition 5.1.1 (Barbs and Contexts)

Predicate $N \Downarrow l$ holds true if and only if $N \equiv (\nu \tilde{l})(N' \parallel l :: \langle t \rangle)$ for some \tilde{l} , N' and t such that $l \notin \tilde{l}$.

Predicate $N \Downarrow l$ holds true if and only if $N \Longrightarrow N'$ for some N' such that $N' \Downarrow l$.

A context $C[\cdot]$ is a μKLAIM net with an occurrence of a hole $[\cdot]$ to be filled in with any net. Formally,

$$C[\cdot] ::= [\cdot] \quad | \quad N \parallel C[\cdot] \quad | \quad (\nu l)C[\cdot]$$

We have chosen the basic observables by taking inspiration from those used for the asynchronous π -calculus [5]. One may wonder if our choice is “correct” and argue that there are other alternative notions of basic observables that seem quite natural, as we have discussed before. A first alternative could be to consider as equivalent two nets if they make available the same set of data, possibly in different nodes (this choice can be defended as follows: when a user looks for a paper in the web he may not be interested whether he finds the paper in author’s web pages or in publisher’s one). A second alternative could be to consider as equivalent two nets if they have exactly the same data at the same localities. Later on, we shall prove that the congruences induced by these alternative observables do coincide. This means that our results are quite independent from the observable chosen and vindicates our choice. Moreover, notice that, by using other kinds of observation predicates, more sophisticated equivalences should come into the picture. For example, in [15] it is shown how *must testing* and *fair testing* can be obtained in CCS by only changing the basic observable.

Now, we say that a binary relation \mathfrak{R} between nets is

- *barb preserving*, if $N \mathfrak{R} M$ and $N \Downarrow l$ imply $M \Downarrow l$;
- *reduction closed*, if $N \mathfrak{R} M$ and $N \mapsto N'$ imply $M \Longrightarrow M'$ and $N' \mathfrak{R} M'$, for some M' ;
- *context closed*, if $N \mathfrak{R} M$ implies $C[N] \mathfrak{R} C[M]$ for every context $C[\cdot]$.

Our touchstone equivalences should at the very least relate nets with the same observable behaviour; thus, they must be barb preserving. However, an equivalence defined only in terms of this property are too weak: indeed, the set of barbs changes during computations or when interacting with an external environment. Moreover, for the sake of compositionality, our touchstone equivalences should also be congruences. These requirements lead us to the following definitions.

Definition 5.1.2 (May testing) \simeq *is the largest symmetric, barb preserving and context closed relation between nets.*

Definition 5.1.3 (Barbed congruence) \cong is the largest symmetric, barb preserving, reduction and context closed relation between nets.

We want to remark that the above definition of barbed congruence is the standard one, see [96, 124]. May testing is, instead, usually defined in terms of *observers*, *experiments* and possible *successes of experiment* [65]. However, if we let \simeq' denote the equivalence on μKLAIM nets defined in [65], we can prove that the two definitions do coincide. Moreover, the inclusions between our touchstone equivalences reflect the inclusions that hold in the π -calculus. To define may testing like in [65], we let `test` be a fresh and reserved name used to report *success* of an *experiment* (i.e. a computation) of a net and an *observer*. The latter is a net containing (i) a node whose address is `test` that can only host the datum $\langle \text{test} \rangle$, and (ii) processes that may emit the datum $\langle \text{test} \rangle$ at `test`. A computation reports success if, along its execution, a datum $\langle \text{test} \rangle$ at node `test` appears; this is written \xrightarrow{OK} .

Definition 5.1.4 $N \simeq' M$ if, for any observer K , it holds that $N \parallel K \xrightarrow{OK}$ if and only if $M \parallel K \xrightarrow{OK}$.

Proposition 5.1.5 $\cong \subseteq \simeq = \simeq'$.

Proof: That \cong is a sub-relation of \simeq trivially follows from their definitions. The inclusion is strict because the latter equivalence abstracts from the branching structure of the equated nets, while the former one does not (because of reduction closure). This is standard in process calculi, e.g., in CCS and in π -calculus.

We start proving that $\simeq \subseteq \simeq'$. Let $N \simeq M$ and pick up any observer K such that $N \parallel K \xrightarrow{OK}$. Then, by contextuality, $N \parallel K \simeq M \parallel K$ and, by barb preservation, $N \parallel K \Downarrow \text{test}$ (that comes from $N \parallel K \xrightarrow{OK}$) implies that $M \parallel K \Downarrow \text{test}$. Since `test` is a name occurring only in K (by definition of observers), it must be $M \parallel K \xrightarrow{OK}$, as required.

Viceversa, we need to prove that \simeq' is barb preserving and context closed. Let $N \simeq' M$.

Barb preservation. Let $N \Downarrow l$, i.e. $N \Vdash (\nu \tilde{l})(N' \parallel l :: \langle t \rangle)$ and consider $K \triangleq \text{test} :: \text{in}(T)@l.\text{out}(\text{test})@\text{test}$, where T is a template matching against t and such that $\text{fn}(T) \cap \tilde{l} = \emptyset$. Then, $N \parallel K \xrightarrow{OK}$ that, by hypothesis, implies $M \parallel K \xrightarrow{OK}$. Now, because of freshness of `test`, this is possible only if $M \Downarrow l$.

Context closure. The proof is by induction on the structure of the context $C[\cdot]$. The base case is trivial. For the inductive case, we have two possibilities:

- $C[\cdot] \triangleq \mathcal{D}[\cdot] \parallel H$. By induction, $\mathcal{D}[N] \simeq' \mathcal{D}[M]$; we pick up an observer K and prove that $C[N] \parallel K \xrightarrow{OK}$ implies $C[M] \parallel K \xrightarrow{OK}$

(by symmetry, this suffices). We now consider the observer $H \parallel K$; by Definition 5.1.4, by induction and by the fact that $\mathcal{D}[N] \parallel (H \parallel K) \xrightarrow{OK}$ we have that $\mathcal{D}[M] \parallel (H \parallel K) \xrightarrow{OK}$. The thesis easily follows by rule (STR-PAss) and because $\equiv \subseteq \simeq'$.

- $C[\cdot] \triangleq (\nu l)\mathcal{D}[\cdot]$. By induction, $\mathcal{D}[N] \simeq' \mathcal{D}[M]$; we pick up an observer K and we prove that $C[N] \parallel K \xrightarrow{OK}$ implies $C[M] \parallel K \xrightarrow{OK}$. Since l is bound, we can assume, up-to alpha-equivalence, that $l \notin \text{fn}(K)$; in particular, $l \neq \text{test}$. Now, $C[N] \parallel K \xrightarrow{OK}$ if and only if $\mathcal{D}[N] \parallel K \xrightarrow{OK}$ (and similarly when replacing N with M). This suffices to conclude. ■

Traditionally [146, 147], barbed congruence and may testing are, respectively, the bottom and the top element of the lattice induced by the subset inclusion between (weak) equivalences on a given process calculus. Practically all the interesting weak equivalences that have been developed in process calculi fall between these two kinds of equivalences. Moreover, a part from the theoretical interest, it is worth to study both these equivalences because the former one enjoys a more effective proof technique, while the latter one often suffices to express properties of programs (mainly, those properties that can be expressed in terms of reachability of a particular state). Hence, we do not have any preference among these two equivalences and we consider both of them as a benchmark for μKLAIM .

The problem behind the definition of barbed congruence and may testing is that context closure makes it hard to prove equivalences due to the universal quantification over contexts. In the following sections, we shall provide two tractable characterisations of these equivalences, as a *bisimulation-based* and as a *trace-based* equivalence.

Before doing this, we show that we can change observables without changing the congruences they induce; this proves the robustness of our touchstone equivalences and supports our choice. Other two reasonable observables in a global computing framework can be existence of a specific (visible) datum in some node of a net, or existence of a specific datum in a specific node of a net.

Definition 5.1.6 (Alternative Touchstone Equivalences) *Let $\cong_1, \cong_2 \simeq_1$ and \simeq_2 be the reduction barbed congruences and the may testing equivalences obtained by replacing the observable of Definition 5.1.1, respectively, with the following ones:*

1. $N \downarrow \langle t \rangle$ iff $N \equiv (\nu \tilde{l})(N' \parallel l :: \langle t \rangle)$ for some l and \tilde{l} such that $\{l, t\} \cap \tilde{l} = \emptyset$
2. $N \downarrow_l \langle t \rangle$ iff $N \equiv (\nu \tilde{l})(N' \parallel l :: \langle t \rangle)$ for some \tilde{l} such that $\{l, t\} \cap \tilde{l} = \emptyset$

We now prove that, thanks to contextuality, \cong_1 and \cong_2 coincide with \cong , and that \simeq_1 and \simeq_2 coincide with \simeq .

Proposition 5.1.7 $\cong_1 = \cong_2 = \cong$ and $\simeq_1 = \simeq_2 = \simeq$.

Proof: Notice that we only need to consider barb preservation. Indeed, contextuality and reduction closure (in the case of barbed congruence) are ensured by definition. We just work with barbed congruences; the proofs for may testing can be then extracted straightforwardly.

$\cong_2 \subseteq \cong_1$. Let $N \cong_2 M$. Suppose that $N \Downarrow \langle t \rangle$. This implies that $\exists l : N \Downarrow_l \langle t \rangle$. Hence, by hypothesis, $M \Downarrow_l \langle t \rangle$ that, by definition, implies $M \Downarrow \langle t \rangle$.

$\cong_1 \subseteq \cong$. Let $N \cong_1 M$ and $N \Downarrow l$, i.e. $N \Vdash (\nu \bar{l})(N' \parallel l :: \langle t \rangle)$. Then $M \Downarrow l$, otherwise the context $[\cdot] \parallel l' :: \mathbf{in}(T) @ l . \mathbf{out}(l') @ l'$ (for l' fresh and T matching against t such that $fn(T) \cap \bar{l} = \emptyset$) would break \cong_1 .

$\cong \subseteq \cong_2$. This case is similar. ■

5.2 Bisimulation Equivalence

To coinductively capture barbed congruence, we introduce a labelled transition system (LTS) to make apparent the action a net is willing to perform in order to evolve. For the sake of presentation, we introduce the syntactic category of *inert components*

$$I ::= \mathbf{nil} \quad | \quad \langle t \rangle$$

for grouping those components that are unable to perform any basic operation. The *labelled transition relation*, $\xrightarrow{\alpha}$, is defined as the least relation over nets induced by the inference rules in Table 3.8. Transition labels take the form

$$\chi ::= \tau \quad | \quad (\nu \bar{l}) I @ l \quad \quad \alpha ::= \chi \quad | \quad \triangleright l \quad | \quad t \triangleleft l$$

We will write $bn(\alpha)$ for \bar{l} if $\alpha = (\nu \bar{l}) I @ l$ and for \emptyset , otherwise; $fn(\alpha)$ is defined accordingly.

Let us now briefly comment on some rules of the LTS; most of them are adapted from the π -calculus [124]. Rule (LTS-EXISTS) signals existence of nodes (label $\mathbf{nil} @ l$) or of data (label $\langle t \rangle @ l$). Rules (LTS-OUT) and (LTS-EVAL) express the intention of spawning a component and require the existence of the target node to complete successfully (rule (LTS-SEND)). Similarly, rules (LTS-IN) (given in an early style) and (LTS-READ) express the intention of performing an input; this input is actually performed (rule (LTS-COMM)) only if the chosen datum is present in the target node. Notice that, in the right hand side of these rules, existence of the node target of the action can be assumed: indeed, if l provides datum $\langle t \rangle$, this implies that l does exist. Rule (LTS-OPEN) signals extrusion of bound names; as in some presentation of the π -calculus, this rule is used to investigate the capability of processes to export bound names, rather than to extend the scope of bound names. To this last aim, law (STR-EXT) is used; in fact, in rule (LTS-COMM) labels do not

<p>(LTS-OUT)</p> $\frac{}{l :: \mathbf{out}(t)@l'.P \xrightarrow{\triangleright l'} l :: P \parallel l' :: \langle t \rangle}$	<p>(LTS-EVAL)</p> $\frac{}{l :: \mathbf{eval}(Q)@l'.P \xrightarrow{\triangleright l'} l :: P \parallel l' :: Q}$
<p>(LTS-IN)</p> $\frac{\text{match}(T, t) = \sigma}{l :: \mathbf{in}(T)@l'.P \xrightarrow{t \triangleleft l'} l :: P\sigma \parallel l' :: \mathbf{nil}}$	<p>(LTS-READ)</p> $\frac{\text{match}(T, t) = \sigma}{l :: \mathbf{read}(T)@l'.P \xrightarrow{t \triangleleft l'} l :: P\sigma \parallel l' :: \langle t \rangle}$
<p>(LTS-NEW)</p> $\frac{}{l :: \mathbf{new}(l').P \xrightarrow{\tau} (v l')(l :: P \parallel l' :: \mathbf{nil})}$	<p>(LTS-EXISTS)</p> $\frac{}{l :: I \xrightarrow{l @ l} l :: \mathbf{nil}}$
<p>(LTS-COMM)</p> $\frac{N_1 \xrightarrow{t \triangleleft l'} N'_1 \quad N_2 \xrightarrow{\langle t \rangle @ l'} N'_2}{N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2}$	<p>(LTS-SEND)</p> $\frac{N_1 \xrightarrow{\triangleright l} N'_1 \quad N_2 \xrightarrow{\mathbf{nil} @ l} N'_2}{N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2}$
<p>(LTS-RES)</p> $\frac{N \xrightarrow{\alpha} N' \quad l \notin n(\alpha)}{(v l)N \xrightarrow{\alpha} (v l)N'}$	<p>(LTS-OPEN)</p> $\frac{N \xrightarrow{(v \bar{l}) \langle t \rangle @ l'} N' \quad l \in \text{fn}(t) - \{\bar{l}, l'\}}{(v l)N \xrightarrow{(v \bar{l}, l) \langle t \rangle @ l'} N'}$
<p>(LTS-PAR)</p> $\frac{N_1 \xrightarrow{\alpha} N_2 \quad \text{bn}(\alpha) \cap \text{fn}(N) = \emptyset}{N_1 \parallel N \xrightarrow{\alpha} N_2 \parallel N}$	<p>(LTS-STRUCT)</p> $\frac{N \equiv M \xrightarrow{\alpha} M' \equiv N'}{N \xrightarrow{\alpha} N'}$

Table 5.1: μKLAIM Labelled Transition System (LTS)

carry any restriction on names, whose scope must have been previously extended. Rules (LTS-RES), (LTS-PAR) and (LTS-STRUCT) are standard.

To simplify the proofs, we consider a slightly extended structural congruence: it is defined like in Table 2.2 (with the obvious adaption of KLAIM nodes to μKLAIM ones) plus the rule

$$\text{(STR-RNODE)} \quad (v l)N \equiv (v l)(N \parallel l :: \mathbf{nil})$$

saying that any restricted name can be used as the address of a node¹. In what follows, we shall only consider nets where each bound name is associated to a node; by virtue of rule (STR-RNODE) this is always possible. This is necessary, otherwise our semantic theories would distinguish names restricted at the outset

¹Restricted names can be thought of as private network addresses, whose corresponding nodes can be activated when needed, and successively deactivated, by the owners of the resource (i.e. the nodes included in the scope of the restriction).

Moreover, notice that we did not include this rule from the beginning (i.e., in Table 2.2) because it would have created problems when dealing with types: the type of a restricted node derives from the type of its creator and rule (STR-RNODE) does not keep this information into account.

from those created during computations: the former ones are not necessarily the address of a node, while the latter ones are (see rules (RED-NEW) and (LTS-NEW)).

Notation 5.2.1 We shall write $N \xrightarrow{\alpha}$ to mean that there exists a net N' such that $N \xrightarrow{\alpha} N'$. Alternatively, we could say that N can perform a α -step. Moreover, we shall usually denote relation composition by juxtaposition; thus, e.g., $N \xrightarrow{\alpha \alpha'} M$ means that there exists a net N' such that $N \xrightarrow{\alpha} N' \xrightarrow{\alpha'} M$. We shall use the convention that putting a bar over a relation means that such a relation does not hold (e.g. $N \not\xrightarrow{\alpha} N'$ means that N cannot reduce to N' performing α). As usual, we let \Rightarrow to stand for $\xrightarrow{\tau^*}$, $\overset{\alpha}{\Rightarrow}$ to stand for $\Rightarrow \xrightarrow{\alpha} \Rightarrow$, and $\overset{\hat{\alpha}}{\Rightarrow}$ to stand for \Rightarrow , if $\alpha = \tau$, and for $\overset{\alpha}{\Rightarrow}$, otherwise.

The LTS we have just defined is ‘correct’ w.r.t. the operational semantics of μKLAIM , as stated by the following Proposition. Notice that \mapsto is the actual semantics of μKLAIM ; the LTS of Table 5.1 can be thought of as a technical device deployed to give a tractable formulation of barbed congruence.

Proposition 5.2.2 $N \mapsto M$ if and only if $N \xrightarrow{\tau} M$.

Proof: Both the directions are proved by an easy induction on the length of the shortest inference leading to the judgements. ■

Now, we prove some relationships between transitions of the LTS and the syntactical form of the net performing them.

Proposition 5.2.3 *The following facts hold:*

1. $N \xrightarrow{\text{nil} @ l} N'$ if and only if $N \equiv N'' \parallel l :: \text{nil}$; moreover, $N'' \equiv N' \equiv N$.
2. $N \xrightarrow{(\widetilde{v}l) \langle t \rangle @ l} N'$ if and only if $N \equiv (\widetilde{v}l)(N'' \parallel l :: \langle t \rangle)$ for $l \notin \widetilde{l}$; moreover, $N' \equiv N'' \parallel l :: \text{nil}$.

Proof: In both statements, the ‘if’ part is straightforward, by using (LTS-EXISTS) and (LTS-STRUCT)/(LTS-PAR)/(LTS-OPEN)/(LTS-RES). For the converse, the first claim is proved by a straightforward induction on the inference. The second claim is proved in a slightly more elaborated way. Indeed, we need a double induction: one on the cardinality of \widetilde{l} , and the other on the inference length. We leave the details to the reader. ■

We now characterise barbed congruence by using the labels of the LTS instead of the universal quantification over all contexts. In this way, we obtain an alternative characterisation of \cong in terms of a labelled *bisimilarity*.

Definition 5.2.4 (Bisimilarity) A symmetric relation \mathfrak{R} between μKLAIM nets is a (weak) bisimulation if for each $N \mathfrak{R} M$ it holds that:

1. if $N \xrightarrow{x} N'$ then, for some M' , $M \xrightarrow{\hat{x}} M'$ and $N' \mathfrak{R} M'$;
2. if $N \xrightarrow{\triangleright l} N'$ then, for some M' , $M \parallel l :: \mathbf{nil} \Rightarrow M'$ and $N' \mathfrak{R} M'$;
3. if $N \xrightarrow{t \triangleleft l} N'$ then, for some M' , $M \parallel l :: \langle t \rangle \Rightarrow M'$ and $N' \mathfrak{R} M'$.

Bisimilarity, \approx , is the largest bisimulation.

Our bisimulation is somehow inspired by that in [119]. The key idea is that, since sending operations are asynchronous, the evolution $N \xrightarrow{\triangleright l} N'$ can be simulated by a net M (in a context where locality l is present) through execution of some internal actions that lead to M' . Indeed, since we want our bisimulation to be a congruence, a context that provides the target locality of the sending action must not tell apart N and M . Hence, for $N \parallel l :: \mathbf{nil}$ to be simulable by $M \parallel l :: \mathbf{nil}$, it must hold that, upon transitions, N' be simulable by M' . Similar considerations hold also for input actions (third item of Definition 5.2.4), but the context now is $[\cdot] \parallel l :: \langle t \rangle$.

Remarkably, though μKLAIM is higher order (processes occur as arguments in process actions, namely in the **eval**), the LTS and the bisimulation we developed do not use labels containing processes. Thus, the bisimulation relies only on a standard quantification over names (in the input case) and we strongly conjecture that it is decidable, under proper assumptions: techniques similar to those in [116] could be used here. Moreover, the presence of rule (LTS-STRUCT) in the LTS does not compromise the tractability of \approx : as standard, (LTS-STRUCT) can be dropped, once we accept to have more rules in the LTS.

5.2.1 Soundness w.r.t. Barbed Congruence

The key result of this subsection is Lemma 5.2.7 that will easily allow us to conclude that bisimilarity is a sound proof technique for barbed congruence. To prove this result, we need some technical tools. First of all, we introduce the notion of *bisimulation up-to structural congruence*: it is defined as a labelled bisimulation except for the fact that the \mathfrak{R} in the consequents of Definition 5.2.4 is replaced by the relation $\equiv \mathfrak{R} \equiv$. Lemma 5.2.5 shows that a bisimulation up-to \equiv is a bisimulation. Then, Lemma 5.2.6 characterises all the possible executions of the net $C[N]$ in terms of the evolutions of N and $C[\cdot]$ separately.

Lemma 5.2.5 *If $N \approx M$ then for any nets N' and M' such that $N \equiv N'$ and $M \equiv M'$ it holds that $N' \approx M'$.*

Proof: Let $\mathfrak{R} \triangleq \{ (N_1, N_2) : N_i \equiv N'_i \text{ and } N'_1 \approx N'_2 \}$. We shall prove that \mathfrak{R} is a labelled bisimulation. Let $N_1 \xrightarrow{\alpha} M_1$; by (LTS-STRUCT) we have that $N'_1 \equiv$

$N_1 \xrightarrow{\alpha} M_1$. We only consider the case for $\alpha = \chi$; the other cases are similar. By hypothesis, $N_2 \xrightarrow{\hat{\chi}} M_2$ for some M_2 such that $M_1 \approx M_2$. Then, $N_2 \equiv N_2' \xrightarrow{\hat{\chi}} M_2$ and $(M_1, M_2) \in \mathfrak{R}$ because of reflexivity of \equiv . ■

Lemma 5.2.6 $C[N] \xrightarrow{\alpha} \bar{N}$ if and only if one of the following conditions hold:

1. $N \xrightarrow{\alpha} N'$ with $n(\alpha) \cap \text{bn}(C[\cdot]) = \emptyset$, or
2. $C[\mathbf{0}] \xrightarrow{\alpha} C'[\mathbf{0}]$, or
3. $N \xrightarrow{(\bar{\nu}l) \langle t \rangle @ l} N'$ with $\alpha = (\bar{\nu}l', \bar{l}) \langle t \rangle @ l$, $C[\cdot] \triangleq C_1[(\bar{\nu}l')C_2[\cdot]]$ and $\text{fn}(\alpha) \cap \text{bn}(C_1[\cdot], C_2[\cdot]) = \emptyset$, or
4. $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{\text{nil} @ l} H'$, $N \xrightarrow{\triangleright l} N'$ and $l \notin \text{bn}(C_2[\cdot])$, or
5. $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{\triangleright l} H'$, $N \xrightarrow{\text{nil} @ l} N'$ and $l \notin \text{bn}(C_2[\cdot])$, or
6. $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{(\bar{\nu}l) \langle t \rangle @ l} H'$, $N \xrightarrow{t \triangleleft l} N'$ and $\{l, t\} \cap \text{bn}(C_2[\cdot]) = \emptyset$, or
7. $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{t \triangleleft l} H'$, $N \xrightarrow{(\bar{\nu}l) \langle t \rangle @ l} N'$ and $l \notin \text{bn}(C_2[\cdot])$

Moreover, the resulting net \bar{N} is, respectively, structurally equivalent to $C[N']$, or $C'[N]$, or $C_1[C_2[N']]$, or $C[N']$, or $C_1[C_2[N] \parallel H']$, or $C_1[(\bar{\nu}l)C_2[N' \parallel H']]$ (cases 6 and 7.). Finally, $\alpha = \tau$ in cases 4., 5., 6., and 7. .

Proof: The “if” part is trivial, by using the LTS of Table 5.1 and by observing that $M \xrightarrow{\alpha} M'$ with $n(\alpha) \cap \text{bn}(\mathcal{D}[\cdot]) = \emptyset$ implies $\mathcal{D}[M] \xrightarrow{\alpha} \mathcal{D}[M']$. The “only if” part is proved by induction on the length of the inference of $\xrightarrow{\alpha}$. In the base case (length 1), it must be $C[\cdot] \triangleq [\cdot]$; hence, obviously $C[N] \triangleq N \xrightarrow{\alpha} N' \triangleq C[N']$ (and hence we fall in case 1. of this Lemma). For the inductive step, we reason by case analysis on the last rule applied in the inference:

(LTS-RES). In this case, it must be

$$\frac{\mathcal{D}[N] \xrightarrow{\alpha} \bar{N}' \quad l \notin n(\alpha)}{C[N] \triangleq (\nu l)\mathcal{D}[N] \xrightarrow{\alpha} (\nu l)\bar{N}'}$$

We can now apply induction and reason by analysis on the used case of this lemma.

1. $N \xrightarrow{\alpha} N'$, $n(\alpha) \cap \text{bn}(\mathcal{D}[\cdot]) = \emptyset$ and $\bar{N}' \equiv \mathcal{D}[N']$. Hence, we still fall in case 1. by using the context $C[\cdot] \triangleq (\nu l)\mathcal{D}[\cdot]$.
2. $\mathcal{D}[\mathbf{0}] \xrightarrow{\alpha} \mathcal{D}'[\mathbf{0}]$ and $\bar{N}' \equiv \mathcal{D}'[N]$. Hence, we still fall in case 2. with contexts $C[\cdot] \triangleq (\nu l)\mathcal{D}[\cdot]$ and $C'[\cdot] \triangleq (\nu l)\mathcal{D}'[\cdot]$.

3. $N \xrightarrow{(\nu \bar{l}) \langle t \rangle @ l'} N'$, $\alpha \triangleq (\nu \bar{l}', \bar{l}) \langle t \rangle @ l'$, $\mathcal{D}[\cdot] \triangleq \mathcal{D}_1[(\nu \bar{l}') \mathcal{D}_2[\cdot]]$ and $n(\alpha) \cap \text{bn}(\mathcal{D}_1[\cdot], \mathcal{D}_2[\cdot]) = \emptyset$; moreover, $\bar{N}' \equiv \mathcal{D}_1[\mathcal{D}_2[N']]$. Hence, we still fall in case 3. by using the contexts $C_1[\cdot] \triangleq (\nu l) \mathcal{D}_1[\cdot]$ and $C_2[\cdot] \triangleq \mathcal{D}_2[\cdot]$.
4. $\mathcal{D}[\cdot] \triangleq \mathcal{D}_1[\mathcal{D}_2[\cdot] \parallel H]$ with $H \xrightarrow{\text{nil} @ l} H'$, $N \xrightarrow{\triangleright l} N'$ and $l \notin \text{bn}(\mathcal{D}_2[\cdot])$; moreover, $\mathcal{D}[N] \xrightarrow{\tau} \mathcal{D}[N']$. Hence we still fall in case 4. by using the contexts $C_1[\cdot] \triangleq (\nu l) \mathcal{D}_1[\cdot]$ and $C_2[\cdot] \triangleq \mathcal{D}_2[\cdot]$.
5. 6. and 7. are similar.

(LTS-OPEN). In this case, it must be

$$\frac{\mathcal{D}[N] \xrightarrow{(\nu \bar{l}) \langle t \rangle @ l} \bar{N} \quad l' \in t - \{\bar{l}, l\}}{C[N] \triangleq (\nu l') \mathcal{D}[N] \xrightarrow{(\nu l', \bar{l}) \langle t \rangle @ l} \bar{N}}$$

We can now apply induction and reason by analysis on the used case of this lemma; we have to consider only the first three cases.

1. $N \xrightarrow{(\nu \bar{l}) \langle t \rangle @ l} N'$, $\{l, t\} \cap \text{bn}(\mathcal{D}[\cdot]) = \emptyset$ and $\bar{N} \triangleq \mathcal{D}[N']$. Hence, we fall in case 3. by using the contexts $C_1[\cdot] \triangleq [\cdot]$ and $C_2[\cdot] \triangleq \mathcal{D}[\cdot]$.
2. $\mathcal{D}[\mathbf{0}] \xrightarrow{(\nu \bar{l}) \langle t \rangle @ l} \mathcal{D}'[\mathbf{0}]$ and $\bar{N} \triangleq \mathcal{D}'[N]$. Hence, we still fall in case 2. with contexts $C[\cdot] \triangleq (\nu l_2) \mathcal{D}[\cdot]$ and $C'[\cdot] \triangleq \mathcal{D}'[\cdot]$.
3. $N \xrightarrow{(\nu \bar{l}_1) \langle t \rangle @ l} N'$, $\mathcal{D}[\cdot] \triangleq \mathcal{D}_1[(\nu \bar{l}_2) \mathcal{D}_2[\cdot]]$, $\bar{l} = \bar{l}_1, \bar{l}_2$ and $\{t, l\} \cap \text{bn}(\mathcal{D}_1[\cdot], \mathcal{D}_2[\cdot]) = \emptyset$; moreover, $\bar{N} \triangleq \mathcal{D}_1[\mathcal{D}_2[N']]$. Hence, we still fall in case 3. by using the contexts $C_1[\cdot] \triangleq [\cdot]$ and $C_2[\cdot] \triangleq \mathcal{D}[\cdot]$.

(LTS-PAR). In this case, one of the following inferences should hold:

$$\frac{K \xrightarrow{\alpha} K' \quad \text{bn}(\alpha) \cap n(\mathcal{D}[N]) = \emptyset}{C[N] \triangleq \mathcal{D}[N] \parallel K \xrightarrow{\alpha} \mathcal{D}[N] \parallel K'} \quad \text{or} \quad \frac{\mathcal{D}[N] \xrightarrow{\alpha} \bar{N}' \quad \text{bn}(\alpha) \cap n(K) = \emptyset}{C[N] \triangleq \mathcal{D}[N] \parallel K \xrightarrow{\alpha} \bar{N}' \parallel K}$$

By using the first inference, we fall in case 2. with resulting context $C'[\cdot] \triangleq \mathcal{D}[\cdot] \parallel K'$. By using the second inference, we can apply inductive arguments similar to those used in the (LTS-RES) case, but now the context $C[\cdot]$ we consider is $\mathcal{D}[\cdot] \parallel K$ instead of $(\nu l) \mathcal{D}[\cdot]$.

(LTS-SEND). In this case, one of the following inferences should hold:

$$\frac{K \xrightarrow{\triangleright l} K' \quad \mathcal{D}[N] \xrightarrow{\text{nil} @ l} \mathcal{D}[N]}{C[N] \triangleq \mathcal{D}[N] \parallel K \xrightarrow{\tau} \mathcal{D}[N] \parallel K'} \quad \text{or} \quad \frac{K \xrightarrow{\text{nil} @ l} K \quad \mathcal{D}[N] \xrightarrow{\triangleright l} \bar{N}'}{C[N] \triangleq \mathcal{D}[N] \parallel K \xrightarrow{\tau} \bar{N}' \parallel K}$$

To apply induction, notice that we only have to consider the first two cases of this lemma, since the actions fired by $\mathcal{D}[N]$ are different from τ and have no restricted names. For the first reduction, we have the following cases.

1. $N \xrightarrow{\text{nil} @ l} N$ and $l \notin \text{bn}(\mathcal{D}[\cdot])$. Hence, we fall in case 5. by using $C_1[\cdot] \triangleq [\cdot]$, $C_2[\cdot] \triangleq \mathcal{D}[\cdot]$ and $H \triangleq K$.
2. $\mathcal{D}[\mathbf{0}] \xrightarrow{\text{nil} @ l} \mathcal{D}[\mathbf{0}]$; hence, we still fall in case 2. with resulting context $C'[\cdot] \triangleq \mathcal{D}[\cdot] \parallel K'$.

For the second reduction, we have similar cases; we just list the differences.

1. $\bar{N}' \triangleq \mathcal{D}[N']$ and we fall into case 4. .
2. $\bar{N}' \triangleq \mathcal{D}'[N]$ and the resulting context is $C'[\cdot] \triangleq \mathcal{D}'[\cdot] \parallel K$.

(LTS-COMM). In this case, one of the following inferences should hold:

$$\frac{\mathcal{D}[N] \xrightarrow{t \triangleleft l} \bar{N}' \quad K \xrightarrow{\langle t \rangle @ l} K'}{C[N] \triangleq \mathcal{D}[N] \parallel K \xrightarrow{\tau} \bar{N}' \parallel K'} \quad \text{or} \quad \frac{K \xrightarrow{t \triangleleft l} K' \quad \mathcal{D}[N] \xrightarrow{\langle t \rangle @ l} \bar{N}'}{C[N] \triangleq \mathcal{D}[N] \parallel K \xrightarrow{\tau} \bar{N}' \parallel K'}$$

Like before, we just have two possible inductive cases for each reduction (namely, the first two of this lemma). For the first reduction we have the following cases.

1. $N \xrightarrow{t \triangleleft l} N'$, $\{l, t\} \cap \text{bn}(\mathcal{D}[\cdot]) = \emptyset$ and $\bar{N}' \triangleq \mathcal{D}[N']$. Hence, we fall in case 6. by using $C_1[\cdot] \triangleq [\cdot]$, $C_2[\cdot] \triangleq \mathcal{D}[\cdot]$ and $H \triangleq K$.
2. $\mathcal{D}[\mathbf{0}] \xrightarrow{t \triangleleft l} \mathcal{D}'[\mathbf{0}]$ and $\bar{N}' \triangleq \mathcal{D}'[N]$. Hence, we still fall in case 2. with resulting context $C'[\cdot] \triangleq \mathcal{D}'[\cdot] \parallel K'$.

For the second reduction, we have similar cases. The only difference is that case 1. in the induction leads to case 7. in the conclusion.

(LTS-STRUCT). In this case, it must be

$$\frac{C[N] \equiv M_1 \xrightarrow{\alpha} M_2 \equiv \bar{N}}{C[N] \xrightarrow{\alpha} \bar{N}}$$

We now proceed by induction on the structure of context $C[\cdot]$. The base case (for $C[\cdot] \triangleq [\cdot]$) trivially falls in case 1. of this Lemma. For the inductive case, let us reason by case analysis on the structure of $C[\cdot]$:

$C[\cdot] \triangleq (\nu l)\mathcal{D}[\cdot]$. We furtherly identify three possible sub-cases:

- if $M_1 \triangleq (\nu l)M$ and $l \in \text{bn}(\alpha)$, for some $M \equiv \mathcal{D}[N]$, then we can apply the structural induction to $\mathcal{D}[N] \xrightarrow{\alpha'} M'$, for some $M' \equiv M_2$ and $\alpha = (\nu l)\alpha'$, and fall in one of the first two cases of this Lemma. By using rule (LTS-OPEN), we can conclude that $C[N] \xrightarrow{\alpha} \bar{N}$ falls in cases 2. or 3. of this Lemma.

- if $M_1 \triangleq (\nu l)M$ and $l \notin \text{bn}(\alpha)$, for some $M \equiv \mathcal{D}[N]$, then we can apply the structural induction to $\mathcal{D}[N] \xrightarrow{\alpha} M'$, for some M' such that $M_2 \equiv (\nu l)M'$, falling in one of the cases of this Lemma. Then, by using (LTS-RES), we can conclude that $C[N] \xrightarrow{\alpha} \bar{N}$ falls in the same case of this Lemma.
- otherwise, we can prove that $C[N] \equiv M'_1 \xrightarrow{\alpha} M_2$ such that $M'_1 \triangleq (\nu l)M$ by using a no longer inference (but possibly using more structural laws). Hence, we can reduce this case to the previous one.

$C[\cdot] \triangleq \mathcal{D}[\cdot] \parallel K$. Because of the structure of $C[\cdot]$, it can be one of the following cases:

- $K \xrightarrow{\alpha} K'$ and $\bar{N} \equiv \mathcal{D}[N] \parallel K'$. In this case, we are trivially in case 2. of this Lemma.
- $\mathcal{D}[N] \xrightarrow{\alpha} \bar{N}'$ and $\bar{N} \equiv \bar{N}' \parallel K$. In this case, we use the structural induction.
- If $\alpha = \tau$ then other four cases are possible:
 - $\mathcal{D}[N] \xrightarrow{\nu l} \bar{N}'$, $K \xrightarrow{\text{nil} @ l} K$ and $\bar{N} \equiv \bar{N}' \parallel K$. By structural induction, it can be that either $N \xrightarrow{\nu l} N'$, or $\mathcal{D}[\mathbf{0}] \xrightarrow{\nu l} \mathcal{D}'[\mathbf{0}]$. In both cases is easy to conclude.
 - $\mathcal{D}[N] \xrightarrow{(\nu \bar{l}) \langle t \rangle @ l} \bar{N}'$, $K \xrightarrow{t \triangleleft l} K'$ and $\bar{N} \equiv (\nu \bar{l})(\bar{N}' \parallel K')$. This case is similar to the previous one.
 - $\mathcal{D}[N] \xrightarrow{\text{nil} @ l} \mathcal{D}[N]$, $K \xrightarrow{\nu l} K'$ and $\bar{N} \equiv \mathcal{D}[N] \parallel K'$. By structural induction, it can be one of the first two cases of this Lemma and we can easily conclude.
 - $\mathcal{D}[N] \xrightarrow{t \triangleleft l} \bar{N}'$, $K \xrightarrow{(\nu \bar{l}) \langle t \rangle @ l} K'$ and $\bar{N} \equiv (\nu \bar{l})(\bar{N}' \parallel K')$. This case is similar to the previous one. ■

Lemma 5.2.7 \approx is a congruence relation.

Proof:

We start by proving that \approx is substitutive w.r.t. to the net contexts $C[\cdot]$, namely that $N \approx M$ implies $C[N] \approx C[M]$ for each $C[\cdot]$. To this aim, we prove that

$$\mathfrak{R} \triangleq \{ (C[N], C[M]) : N \approx M \}$$

is a bisimulation up-to \equiv . Let $C[N] \xrightarrow{\alpha} \bar{N}$; according to Lemma 5.2.6 we have to examine seven cases.

1. $N \xrightarrow{\alpha} N'$ for $n(\alpha) \cap \text{bn}(C[\cdot]) = \emptyset$; we reason by case analysis on α .

$\alpha = \chi$. By hypothesis of bisimilarity, $M \xrightarrow{\hat{\chi}} M'$ and $N' \approx M'$. Hence, trivially, $C[M] \xrightarrow{\hat{\chi}} C[M']$ and, by definition of \mathfrak{R} , it holds that $C[N'] \mathfrak{R} C[M']$.

$\alpha = \triangleright l$. By hypothesis of bisimilarity, $M \parallel l :: \mathbf{nil} \Rightarrow M'$ and $N' \approx M'$. Since $l \notin \text{bn}(C[\cdot])$, we have that $C[M] \parallel l :: \mathbf{nil} \equiv C[M \parallel l :: \mathbf{nil}] \Rightarrow C[M']$ and $C[N'] \mathfrak{R} C[M']$ up-to structural equivalence.

$\alpha = t \triangleleft l$. By hypothesis, $\{l, t\} \cap \text{bn}(C[\cdot]) = \emptyset$; thus, $C[M] \parallel l :: \langle t \rangle \equiv C[M \parallel l :: \langle t \rangle]$. By hypothesis of bisimilarity, $M \parallel l :: \langle t \rangle \Rightarrow M'$ and $N' \approx M'$. Thus, $C[M] \parallel l_1 :: \langle l_2 \rangle \Rightarrow C[M']$ and $C[N'] \mathfrak{R} C[M']$ up-to \equiv .

2. $C[\mathbf{0}] \xrightarrow{\alpha} C'[\mathbf{0}]$; trivially, $C[M] \xrightarrow{\alpha} C'[M]$. Moreover, by definition of \mathfrak{R} , we have that

- if $\alpha = \chi$ then $C'[N] \mathfrak{R} C'[M]$.
- if $\alpha = \triangleright l$, then $C[M] \parallel l :: \mathbf{nil} \Rightarrow C'[M]$ and $C'[N] \mathfrak{R} C'[M]$.
- if $\alpha = t \triangleleft l$, then $C[M] \parallel l :: \langle t \rangle \Rightarrow C'[M]$ and $C'[N] \mathfrak{R} C'[M]$

3. $N \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} N'$, $\alpha = (\widetilde{v\bar{l}'}, \widetilde{l}) \langle t \rangle @ l$, $C[\cdot] \triangleq C_1[(\widetilde{v\bar{l}'})C_2[\cdot]]$, for $n(\alpha) \cap \text{bn}(C_1[\cdot], C_2[\cdot])$, and $\bar{N} \triangleq C_1[C_2[N']]$. By hypothesis of bisimilarity, $M \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} M'$ and $N' \approx M'$; thus, $C[M] \xrightarrow{\alpha} C_1[C_2[M']]$. The thesis easily follows.

4. By hypothesis, $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{\mathbf{nil} @ l} H'$, $N \xrightarrow{\triangleright l} N'$, for $l \notin \text{bn}(C_2[\cdot])$, and $\bar{N} \equiv C[N']$. By Proposition 5.2.3, $H \equiv H' \equiv H \parallel l :: \mathbf{nil}$; thus, $C[M] \equiv C[M \parallel l :: \mathbf{nil}]$. By hypothesis of bisimilarity, $M \parallel l :: \mathbf{nil} \Rightarrow M'$ and $N' \approx M'$. Hence, $C[M] \Rightarrow C[M']$ and $\bar{N} \mathfrak{R} M'$ up-to \equiv .

5. $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{\triangleright l} H'$, $N \xrightarrow{\mathbf{nil} @ l} N'$ and $l \notin \text{bn}(C_2[\cdot])$. By hypothesis of bisimilarity, $M \xrightarrow{\mathbf{nil} @ l} M$ and $N' \approx M'$. Hence, $C[M] \Rightarrow C_1[C_2[M] \parallel H']$. The thesis easily follows.

6. $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} H'$, $N \xrightarrow{t \triangleleft l} N'$, for $\{l, t\} \cap \text{bn}(C_2[\cdot]) = \emptyset$, and $\bar{N} \equiv C_1[(\widetilde{v\bar{l}})(C_2[N'] \parallel H')]$. By Proposition 5.2.3, $H \equiv (\widetilde{v\bar{l}})(H'' \parallel l :: \langle t \rangle)$ and $H' \equiv H'' \parallel l :: \mathbf{nil}$; thus, $C[M] \equiv C_1[(\widetilde{v\bar{l}})(C_2[M \parallel l :: \langle t \rangle] \parallel H'')]$. By hypothesis of bisimilarity, $M \parallel l :: \langle t \rangle \Rightarrow M'$ and $N' \approx M'$. Hence $C[M] \Rightarrow C_1[(\widetilde{v\bar{l}})(C_2[M'] \parallel H'')]$ and $\bar{N} \mathfrak{R} C_1[(\widetilde{v\bar{l}})(C_2[M'] \parallel H')]$ up-to \equiv .

7. $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ with $H \xrightarrow{t \triangleleft l} H'$, $N \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} N'$, for $\{l, t\} \cap \text{bn}(C_2[\cdot]) = \emptyset$, and $\bar{N} \equiv C_1[(\widetilde{v\bar{l}})(C_2[N'] \parallel H')]$. By hypothesis of bisimilarity, $M \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} M'$ and $N' \approx M'$. Hence, $C[M] \Rightarrow C_1[(\widetilde{v\bar{l}})C_2[M' \parallel H']]$ and $\bar{N} \mathfrak{R} C_1[(\widetilde{v\bar{l}})(C_2[M'] \parallel H')]$ up-to \equiv .

We are left with proving that \approx is an equivalence relation. Reflexivity and symmetry follow by definition. To prove transitivity, we consider the relation $\mathfrak{R} \triangleq$

$\{(N_1, N_2) : N_1 \approx N_2\}$ and prove that it is a bisimulation. Let $N_1 \approx M \approx N_2$, $N_1 \xrightarrow{\alpha} N'_1$ and let us reason by case analysis on α :

$\alpha = \chi$. In this case, $M \xRightarrow{\hat{\chi}} M'$ for some M' such that $N'_1 \approx M'$. If $M' \equiv M$, then we conclude up-to \equiv . Otherwise, it must be that $N_2 \xRightarrow{\hat{\chi}} N'_2$ and $M' \approx N'_2$; hence $N'_1 \approx N'_2$ and $N'_1 \not\approx N'_2$.

$\alpha = \triangleright l$. By hypothesis, $M \parallel l :: \mathbf{nil} \Rightarrow M'$ and $N'_1 \approx M'$. By context closure, we have that $M \parallel l :: \mathbf{nil} \approx N_2 \parallel l :: \mathbf{nil}$; hence, $N_2 \parallel l :: \mathbf{nil} \Rightarrow N'_2$ and $M' \approx N'_2$. It is easy to conclude that $N'_1 \not\approx N'_2$.

$\alpha = t \triangleleft l$. Similarly, $M \parallel l :: \langle t \rangle \Rightarrow M'$ and $N'_1 \approx M'$. Moreover, $N_2 \parallel l :: \langle t \rangle \Rightarrow N'_2$ and $M' \approx N'_2$, that implies $N'_1 \not\approx N'_2$. ■

Theorem 5.2.8 (Soundness of \approx w.r.t. \cong) *If $N \approx M$ then $N \cong M$.*

Proof: We shall now prove that \approx is barb preserving, reduction closed and contextual. By definition, this implies that $\approx \subseteq \cong$.

- If $N \Downarrow l$ then $N \xRightarrow{(\tilde{v}\tilde{l}) \langle t \rangle @ l}$, for some t and \tilde{l} such that $l \notin \tilde{l}$; hence, by hypothesis of bisimilarity, $M \xRightarrow{(\tilde{v}\tilde{l}) \langle t \rangle @ l}$ and thus $M \Downarrow l$ (these implications rely on Proposition 5.2.3 and Definition 5.2.4).
- By Proposition 5.2.2, $N \mapsto N'$ implies that $N \xrightarrow{\tau} N'$; this, in turn, implies, by hypothesis of bisimilarity, that $M \Rightarrow M'$ (and, again, by Proposition 5.2.2 this means $M \mapsto M'$) and $N' \approx M'$.
- By Lemma 5.2.7, it holds that $C[N] \approx C[M]$, for all net contexts $C[\cdot]$. ■

5.2.2 Completeness w.r.t. Barbed Congruence

We now want to prove the converse, namely that all barbed congruent processes are also bisimilar. To this aim, we need three technical results. The first one gives some simple equations that hold true w.r.t. barbed congruence. The second result gives an alternative characterisation of the contextuality property of Definition 5.1.3. The third result states that we can throw away fresh localities hosting restricted data without breaking barbed congruence.

Proposition 5.2.9 *The following facts hold:*

1. $(\nu l')(l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle) \cong (\nu l')(l :: P\sigma)$, whenever $\text{match}(T, t) = \sigma$
2. $l :: \mathbf{out}(t)@l'.P \parallel l' :: \mathbf{nil} \cong l :: P \parallel l' :: \langle t \rangle$
3. $l :: \mathbf{eval}(Q)@l'.P \parallel l' :: \mathbf{nil} \cong l :: P \parallel l' :: Q$
4. $(\nu l)N \cong N$ whenever $l \notin \text{fn}(N)$

Proof: The first three equations can be easily proved by providing a proper bisimulation containing each of them; this fact, together with Theorem 5.2.8, proves part (1)/(2)/(3). The last equation is proved by first observing that $\equiv \subseteq \cong$ (this can be easily proved). Then, $(\nu l)N \equiv (\nu l)(N \parallel l :: \mathbf{nil}) \equiv N \parallel (\nu l)(l :: \mathbf{nil}) \cong N \parallel \mathbf{0} \equiv N$ (indeed, $(\nu l)(l :: \mathbf{nil}) \cong \mathbf{0}$, as it can be easily verified). Thus, by inclusion of \equiv in \cong and by transitivity of \cong , the claim holds. ■

Lemma 5.2.10 *A relation \mathfrak{R} is contextual if and only if*

1. $N \mathfrak{R} M$ implies that $N \parallel l :: P \mathfrak{R} M \parallel l :: P$ for any name l and process P , and
2. $N \mathfrak{R} M$ implies that $(\nu l)N \mathfrak{R} (\nu l)M$ for any name l

Proof: It is trivial to prove that contextuality implies points (1) and (2) of this Lemma. For the converse, let us assume $N \mathfrak{R} M$ and pick up a context $C[\cdot]$. We now proceed by induction on the structure of $C[\cdot]$. The base case is trivial. For the inductive case, we identify two possibilities:

1. $C[\cdot] \triangleq \mathcal{D}[\cdot] \parallel K$. By induction, we have that $\mathcal{D}[N] \mathfrak{R} \mathcal{D}[M]$. We now proceed by induction on the structure of K . The base case is trivial, up-to \equiv . For the inductive case, it can be either $K \triangleq l :: P \parallel K'$ or $K \triangleq (\nu l)K'$. In the first case, by point (1) of this Lemma, it holds that $\mathcal{D}[N] \parallel l :: P \mathfrak{R} \mathcal{D}[M] \parallel l :: P$; then, by second induction, we can conclude $C[N] \mathfrak{R} C[M]$. In the second case, we can always assume that l is fresh for $\mathcal{D}[\cdot]$, N and M (this is always possible, up-to alpha-equivalence). By second induction, we have that $\mathcal{D}[N] \parallel K' \mathfrak{R} \mathcal{D}[M] \parallel K'$ and, by point (2) of this Lemma, we can conclude up-to \equiv .
2. $C[\cdot] \triangleq (\nu l)\mathcal{D}[\cdot]$. By induction we have that $\mathcal{D}[N] \mathfrak{R} \mathcal{D}[M]$; thus, by point (2) of this Lemma, it holds that $(\nu l)\mathcal{D}[N] \mathfrak{R} (\nu l)\mathcal{D}[M]$, i.e. $C[N] \mathfrak{R} C[M]$. ■

Lemma 5.2.11 *If $(\nu l)(N \parallel l_f :: \langle l \rangle) \cong (\nu l)(M \parallel l_f :: \langle l \rangle)$ and l_f is fresh for N and M , then $N \cong M$.*

Proof: It suffices to prove that

$$\mathfrak{R} \triangleq \{ (N, M) : l_f \notin n(N, M) \wedge (\nu l)(N \parallel l_f :: \langle l \rangle) \cong (\nu l)(M \parallel l_f :: \langle l \rangle) \}$$

is barb preserving, reduction closed and contextual. Let $N \mathfrak{R} M$.

Barb preservation. Let $N \Downarrow l'$ for $l' \neq l$. Then, it trivially holds that $(\nu l)(N \parallel l_f :: \langle l \rangle) \Downarrow l'$, $(\nu l)(M \parallel l_f :: \langle l \rangle) \Downarrow l'$ and $M \Downarrow l'$ (indeed, $l' \neq l_f$ because of freshness of l_f).

Now, let $N \Downarrow l$, i.e. $N \xrightarrow{(\bar{\nu}l) \langle t \rangle @ l}$. We can consider the context $C[\cdot] \triangleq [\cdot] \parallel l'_f :: \mathbf{in}(!x)@l'_f.\mathbf{in}(T)@x.\mathbf{out}()@l'_f$, where l'_f is fresh and T has been obtained from t by replacing therein each name of \bar{l} with $!y$ (for some \bar{y} fresh). Now, $C[(\nu l)(N \parallel l_f :: \langle l \rangle)] \Downarrow l'_f$; hence, by hypothesis, $C[(\nu l)(M \parallel l_f :: \langle l \rangle)] \Downarrow l'_f$. Because of freshness of l_f and l'_f , it must be that $M \Downarrow l$.

Reduction closure. Let $N \mapsto N'$; thus, $(\nu l)(N \parallel l_f :: \langle l \rangle) \mapsto (\nu l)(N' \parallel l_f :: \langle l \rangle)$. By hypothesis, this fact implies that $(\nu l)(M \parallel l_f :: \langle l \rangle) \mapsto \bar{M}$ such that $(\nu l)(N' \parallel l_f :: \langle l \rangle) \cong \bar{M}$. Since $l_f \notin n(M)$, $l_f :: \langle l \rangle$ is not involved in the transition; thus it follows that $M \mapsto M'$ and $\bar{M} \equiv (\nu l)(M' \parallel l_f :: \langle l \rangle)$. Thus, the claim is proved up-to \equiv .

Contextuality. According to Lemma 5.2.10, we have to prove just two cases.

1. For any l' and P , it holds that $N \parallel l' :: P \mathfrak{R} M \parallel l' :: P$.

Let us first assume that $l \notin fn(l' :: P)$. In this case, it is easy to conclude because $(\nu l)(N \parallel l_f :: \langle l \rangle) \parallel l' :: P \equiv (\nu l)(N \parallel l' :: P \parallel l_f :: \langle l \rangle)$ (and similarly when replacing N with M), by transitivity of \cong and because $\equiv \subseteq \cong$.

We now consider the case in which $l \in fn(P)$ but $l' \neq l$. We use the context $C[\cdot] \triangleq (\nu l_f)([\cdot] \parallel l' :: \mathbf{in}(!x)@l_f.\mathbf{out}(x)@l'_f.P[x/l] \parallel l'_f :: \mathbf{nil})$, where l'_f is a fresh name. Then, $C[(\nu l)(N \parallel l_f :: \langle l \rangle)] \cong (\nu l_f, l)(N \parallel l' :: \mathbf{out}(l)@l'_f.P \parallel l'_f :: \mathbf{nil}) \cong (\nu l_f, l)(N \parallel l' :: P \parallel l'_f :: \langle l \rangle) \cong (\nu l)(N \parallel l' :: P \parallel l'_f :: \langle l \rangle)$. These equalities hold true by using, resp., Proposition 5.2.9(1), (2) and (4). Similarly, $C[(\nu l)(M \parallel l_f :: \langle l \rangle)] \cong (\nu l)(M \parallel l' :: P \parallel l'_f :: \langle l \rangle)$. By transitivity, $C[(\nu l)(N \parallel l_f :: \langle l \rangle)] \cong C[(\nu l)(M \parallel l_f :: \langle l \rangle)]$ implies that $(\nu l)(N \parallel l' :: P \parallel l'_f :: \langle l \rangle) \cong (\nu l)(M \parallel l' :: P \parallel l'_f :: \langle l \rangle)$ and thus $N \parallel l' :: P \mathfrak{R} M \parallel l' :: P$.

The case for $l' = l$ is dealt with similarly. It uses context $C[\cdot] \triangleq (\nu l_f)([\cdot] \parallel l'_f :: \mathbf{in}(!x)@l_f.\mathbf{eval}(P[x/l])@x.\mathbf{out}(x)@l'_f)$, where l'_f is a fresh name, and Proposition 5.2.9(3).

2. For any l' , it holds that $(\nu l')N \mathfrak{R} (\nu l')M$.

Let $l' \neq l, l_f$. Then $(\nu l', l)(N \parallel l_f :: \langle l \rangle) \equiv (\nu l)((\nu l')N \parallel l_f :: \langle l \rangle)$ (and similarly when replacing N with M). By transitivity of \cong and because $\equiv \subseteq \cong$ we can easily conclude.

Let $l' = l \neq l_f$. Then we consider the context $C[\cdot] \triangleq (\nu l_f)([\cdot] \parallel l'_f :: \mathbf{in}(!x)@l_f.\mathbf{new}(l'').\mathbf{out}(l'')@l'_f)$, for l'_f and l'' fresh. Then $C[(\nu l)(N \parallel l_f :: \langle l \rangle)] \cong (\nu l'')(N \parallel l'_f :: \langle l \rangle)$ (and similarly when replacing N with M). Thus, we can conclude that $(\nu l'')(N \parallel l'_f :: \langle l \rangle) \cong (\nu l'')(M \parallel l'_f :: \langle l \rangle)$ that implies $(\nu l)N \mathfrak{R} (\nu l)M$.

Finally, let $l' = l_f \neq l$. Then we consider the context $C[\cdot] \triangleq (\nu l_f)([\cdot] \parallel l'_f :: \mathbf{in}(!x)@l_f.\mathbf{out}(x)@l'_f)$, for l'_f fresh. Then $C[(\nu l)(N \parallel l_f :: \langle l \rangle)] \cong (\nu l)((\nu l_f)N \parallel l'_f :: \langle l \rangle)$ (and similarly when replacing N with M). Thus, we can conclude. \blacksquare

Theorem 5.2.12 (Completeness of \approx w.r.t. \cong) *If $N \cong M$ then $N \approx M$.*

Proof: It is enough to prove that \cong is a bisimulation. We now pick up a transition $N \xrightarrow{\alpha} N'$ and reason by case analysis on α .

$\alpha = \tau$. This case is simple because of reduction closure.

$\alpha = (\nu \tilde{l}) \langle t \rangle @ l$. Let $t = l_1, \dots, l_n$ and $\tilde{l} = \{l'_1, \dots, l'_k\}$; then, we pick up k fresh names $\{x_1, \dots, x_k\}$ and we turn t into a template T where each l'_i in t is replaced with $!x_i$ in T . Now, consider the context

$$C[\cdot] \triangleq [\cdot] \parallel l :: \mathbf{in}(T) @ l . \mathit{Produce}_{l'_f}^1 \dots \mathit{Produce}_{l'_f}^n \parallel \prod_{i=1}^n l'_f \ :: \mathbf{nil}$$

with the l'_f s fresh and

$$\mathit{Produce}_{l'_f}^i \triangleq \begin{cases} \mathbf{new}(l'_i) . \mathbf{out}(l'_i) @ l'_f & \text{if } l_i \notin \tilde{l} \\ \mathbf{out}(l_i) @ l'_f & \text{otherwise} \end{cases}$$

where the l'_i s are all different (fresh) names. Hence, we have that

$$C[N] \Longrightarrow (\nu \hat{l}_1, \dots, \hat{l}_n) (N' \parallel \prod_{i=1}^n l'_f \ :: \langle \hat{l}_i \rangle) \triangleq \hat{N}$$

where

$$\hat{l}_i \triangleq \begin{cases} l'_i & \text{if } l_i \notin \tilde{l} \\ l_i & \text{otherwise} \end{cases}$$

By contextuality and reduction closure, it must be that $C[M] \Longrightarrow \hat{M}$ such that $\hat{N} \cong \hat{M}$. This implies that $\hat{M} \Downarrow l'_f$, for each i ; but this is possible only if $M \xrightarrow{(\nu \tilde{l}) \langle t' \rangle @ l} M'$, for some t' matching T . We are left with proving that $(\nu \tilde{l}') t' =_\alpha (\nu \tilde{l}) t$ and that $N' \cong M'$.

By definition of T , each $l_i \notin \tilde{l}$ is the i -th field of T ; thus, it must also be the i -th field of t' . We have two sub-cases:

- $\tilde{l} = \emptyset$: this case is simple. Indeed, we have that $(\nu \tilde{l}') t' = t' = t$; moreover, because of freshness of the l'_f s and of the l'_i s, we can state that $\hat{M} \equiv (\nu l'_1, \dots, l'_n) (M' \parallel \prod_{i=1}^n l'_f \ :: \langle l'_i \rangle)$. Thus, $M \xrightarrow{\langle t \rangle @ l} M'$ and, by Lemma 5.2.11 iterated n times, $N' \cong M'$.
- $\tilde{l} \neq \emptyset$: we now prove that the remaining fields of t' must be bound. Let $l_i \in \tilde{l}$ and, by contradiction, assume that the i -th field of t' , say \bar{l} , is free. By definition of $\mathit{Produce}_{l'_f}^i$, we have that $\hat{l}_i = l_i$ in \hat{N} . This implies that l'_f hosts in \hat{M} the (free) datum $\langle \bar{l} \rangle$. But then the context

$$[\cdot] \parallel l_f \ :: \mathbf{in}(\bar{l}) @ l'_f . \mathbf{out}() @ l_f$$

for l_f fresh, would falsify $\hat{N} \cong \hat{M}$. Contradiction.

To conclude, we can rename the \tilde{l}' to \tilde{l} (this is possible since we are assuming that bound names are pairwise distinct and different from the

free ones, and hence $\tilde{l} \cap n(M) = \emptyset$. Thus, $M \xrightarrow{(\tilde{v}\bar{l}) \langle t \rangle @ l} M''$ and $\hat{N} \cong (v\hat{l}_1, \dots, \hat{l}_n)(M'' \parallel \prod_{i=1}^n l'_i :: \langle \hat{l}_i \rangle)$; by Lemma 5.2.11 iterated n times, we obtain $N' \cong M''$, as required.

$\alpha = \mathbf{nil} @ l$. We now consider the context $C[\cdot] \triangleq [\cdot] \parallel l_f :: \mathbf{eval}(\mathbf{nil})@l.\mathbf{new}(l')$. $\mathbf{out}(l')@l_f$, for l_f fresh, and the reduction $C[N] \Longrightarrow (v'l')(N' \parallel l_f :: \langle l' \rangle)$. Like before, we have that $M \xrightarrow{\mathbf{nil} @ l} M'$ and $(v'l')(N' \parallel l_f :: \langle l' \rangle) \cong (v'l')(M' \parallel l_f :: \langle l' \rangle)$ that suffices to conclude.

$\alpha = \triangleright l$. We consider the context $[\cdot] \parallel l :: \mathbf{nil}$ and the reduction $N \parallel l :: \mathbf{nil} \mapsto N'$. Then, by contextuality and reduction closure, $M \parallel l :: \mathbf{nil} \Longrightarrow M'$ and $N' \cong M'$. This suffices to conclude (see Definition 5.2.4.2).

$\alpha = t \triangleleft l$. This case is similar to the previous one; we just consider the context $[\cdot] \parallel l :: \langle t \rangle$ and the reduction $N \parallel l :: \langle t \rangle \mapsto N'$. ■

Corollary 5.2.13 (Tractable Characterisation of Barbed Congruence) $\approx = \cong$.

5.3 Trace Equivalence

In this section, we develop a tractable characterisation of may testing. For some well-known process calculi, may testing coincides with trace equivalence [65, 14]; in this section, we show how a similar result is obtained also in the setting of μKLAIM . To the best of our knowledge, this is the first tractable characterisation of may testing for a distributed language with process mobility.

The idea behind trace equivalence is that N and M are related if and only if the sets of their traces coincide. Put in another form, if N exhibits a sequence of visible actions ϱ , then M must exhibit ϱ as well, and viceversa. In an asynchronous setting [16, 48], this requirement must be properly weakened, since the discriminating power of asynchronous contexts is weaker: in the asynchronous π -calculus, for example, contexts cannot observe input actions.

To define a proper trace equivalence we slightly modify the LTS of Table 5.1 by adding the rule

$$(LTS\text{-Rcv}) \quad \frac{N \xrightarrow{t \triangleleft l} N' \quad \tilde{l} = t - fn(N)}{N \xrightarrow{(\tilde{v}\bar{l}) t \triangleleft l} N' \parallel \prod_{l' \in \tilde{l}} l' :: \mathbf{nil}}$$

that permits distinguishing the reception of a free name from the reception of a bound name (this is akin to the asynchronous π -calculus in [16]). In the latter case, the received bound names \tilde{l} must be fresh for the receiving net and, because of law $\underline{\text{STR-RNODE}}$, they can be considered as addresses of nodes; of course, $bn((\tilde{v}\bar{l}) t \triangleleft l) = \tilde{l}$. Notice that rule (LTS-Rcv) is not needed by the bisimulation

we introduced in the previous section to capture barbed congruence. Thus, the new transition system exploits the following labels:

$$\mu ::= \tau \mid \phi \quad \phi ::= (\widetilde{v}\bar{l})I @ l \mid \triangleright l \mid (\widetilde{v}\bar{l})t \triangleleft l$$

where ϕ collects together all the visible labels. Clearly, rules (LTS-RES), (LTS-PAR) and (LTS-STRUCT) from Table 5.1 must now exploit μ instead of α . Then, we define a complementation function over the visible labels of the LTS. Formally,

$$\begin{aligned} \overline{\triangleright \bar{l}} &= \mathbf{nil} @ l & \overline{\mathbf{nil} @ \bar{l}} &= \triangleright l \\ \overline{(\widetilde{v}\bar{l})t \triangleleft l} &= (\widetilde{v}\bar{l})\langle t \rangle @ l & \overline{(\widetilde{v}\bar{l})\langle t \rangle @ l} &= (\widetilde{v}\bar{l})t \triangleleft l \end{aligned}$$

We let ϱ to range over (possibly empty) sequences of visible actions, i.e.

$$\varrho ::= \epsilon \mid \phi \cdot \varrho$$

where ϵ denotes the empty sequence of actions and ‘ \cdot ’ represents concatenation. As usual, $N \xRightarrow{\epsilon}$ denotes $N \Rightarrow$ and $N \xRightarrow{\phi \cdot \varrho}$ denotes $N \xRightarrow{\phi} \xRightarrow{\varrho}$. A naive formulation of trace equivalence such as “ $N \xRightarrow{\varrho}$ if and only if $M \xRightarrow{\varrho}$ ” would be too strong in an asynchronous setting: for example, it would distinguish $N \triangleq l :: \mathbf{in}(!x)@l_1.\mathbf{in}(!y)@l_2$ and $M \triangleq l :: \mathbf{in}(!y)@l_2.\mathbf{in}(!x)@l_1$, which are instead may testing equivalent. Like in [16], a weaker trace equivalence can be defined as follows.

Definition 5.3.1 (Trace Equivalence) \asymp is the largest symmetric relation between μ KLAIM nets such that, whenever $N \asymp M$, it holds that $N \xRightarrow{\varrho}$ implies $M \xRightarrow{\varrho'}$, for some $\varrho' \leq \varrho$.

The crux is to identify a proper ordering on the traces such that may testing is exactly captured by \asymp . The ordering \leq is obtained as the reflexive and transitive closure of the ordering \leq_0 defined in Table 5.2. The intuition behind $\varrho' \leq \varrho$ is that, if a context can interact with a net that exhibits ϱ , then the context can interact with any net that exhibits ϱ' . The ordering \leq_0 relies on the function $(\widetilde{v}\bar{l})\varrho$, that is used in laws (L1), (L2) and (L3) when moving/removing a label of the form $(\widetilde{v}\bar{l})t \triangleleft l$. In this case, the information that \widetilde{l} are fresh received names must be kept in the remaining trace. The formal definition is by induction on \widetilde{l} :

$$\begin{aligned} (\widetilde{v}\emptyset)\varrho &\triangleq \varrho \\ (\widetilde{v}l')\varrho &\triangleq \begin{cases} \varrho & \text{if } l' \notin \mathit{fn}(\varrho) \\ \varrho_1 \cdot (\widetilde{v}l')t' \triangleleft l'' \cdot \varrho_2 & \text{if } \varrho = \varrho_1 \cdot (\widetilde{v}l')t' \triangleleft l'' \cdot \varrho_2 \\ & \text{and } l' \in t' - \mathit{fn}(\varrho_1, l'', \widetilde{l}') \\ \perp & \text{otherwise} \end{cases} \\ (\widetilde{v}l_1, l_2, \dots, l_n)\varrho &\triangleq (\widetilde{v}l_1)((\widetilde{v}l_2, \dots, l_n)\varrho) \end{aligned}$$

(L1)	$\varrho \cdot (\widetilde{v\bar{l}})\varrho' \leq_0 \varrho \cdot (\widetilde{v\bar{l}}) \bullet \Delta l \cdot \varrho'$	if $(\widetilde{v\bar{l}})\varrho' \neq \perp$
(L2)	$\varrho \cdot (\widetilde{v\bar{l}})(\phi \cdot \bullet \Delta l \cdot \varrho') \leq_0 \varrho \cdot (\widetilde{v\bar{l}}) \bullet \Delta l \cdot \phi \cdot \varrho'$	if $(\widetilde{v\bar{l}})(\phi \cdot \bullet \Delta l \cdot \varrho') \neq \perp$
(L3)	$\varrho \cdot (\widetilde{v\bar{l}})\varrho' \leq_0 \varrho \cdot (\widetilde{v\bar{l}}) \bullet \Delta l \cdot \overline{\bullet \Delta l} \cdot \varrho'$	if $(\widetilde{v\bar{l}})\varrho' \neq \perp$
(L4)	$\varrho \cdot \triangleright l \cdot (\widetilde{v\bar{l}}) \bullet \Delta l \cdot \varrho' \leq_0 \varrho \cdot (\widetilde{v\bar{l}}) \bullet \Delta l \cdot \varrho'$	
(L5)	$\varrho \cdot \triangleright l' \cdot (\widetilde{v\bar{l}}) I @ l \cdot \varrho' \leq_0 \varrho \cdot (\widetilde{v\bar{l}}) I @ l \cdot \triangleright l' \cdot \varrho'$	if $l' \notin \widetilde{l}$
where in laws (L1), (L2), (L3) and (L4), $\bullet \Delta l$ stands for either $\triangleright l$ or $t \triangleleft l$ (and hence $\overline{\bullet \Delta l}$ in law (L3) stands for either $\mathbf{nil} @ l$ or $\langle t \rangle @ l$, resp.)		

Table 5.2: The Ordering Relation on Traces

To better understand the motivations underlying this definition, consider the following example that justifies the side condition of law (L1) (similar arguments also hold for laws (L2) and (L3)). In the trace $(v\bar{l}) l' \triangleleft l \cdot \langle l' \rangle @ l''$ performed by a net N , the input action cannot be erased. Indeed, since l' is fresh (see the meaning of label $(v\bar{l}) l' \triangleleft l$), N cannot get knowledge of l' without performing the input and, consequently, cannot perform the action $\langle l' \rangle @ l''$. On the other hand, if N could receive l' from a communication with another node l''' (thus, it can perform action $l' \triangleleft l'''$ after $l' \triangleleft l$), then the first input action can be erased and $(v\bar{l}) l' \triangleleft l''' \cdot \langle l' \rangle @ l'' \leq_0 (v\bar{l}) l' \triangleleft l \cdot l' \triangleleft l''' \cdot \langle l' \rangle @ l''$.

The intuition behind the rules in Table 5.2 now follows. The first three laws have been inspired by [16], while the last three ones are strictly related to the difference between a ‘pure’ name and a name that is used as a node address. Law (L1) states that an input, an output or a migration cannot be directly observed; at most, the *effect* of an output can be observed (by accessing the datum produced by the output). Law (L2) states that the execution of an input/output/migration can be delayed along computations without being noticed by any observer. Law (L3) states that two adjacent ‘complementary’ actions can be deleted (by using a terminology borrowed from CCS [110], we say that ϕ and ϕ' are complementary if they can synchronise to yield a τ – see rules (LTS-COMM) and (LTS-SEND)). Law (L4) states that an input from l always enables outputs/migrations to l ; indeed, if a datum from l has been retrieved, then l exists and any output/migration to it is enabled. Of course, an output/migration to l always enables other outputs/migrations to l . Similarly, law (L5) states that, if an output/migration to l' is enabled after an action ϕ of the form $(\widetilde{v\bar{l}}) I @ l$, then the output/migration can be fired before ϕ , since l' was already present. However, this is not possible if l' has been created before ϕ and ϕ extruded it (i.e., if $l' \in \widetilde{l}$).

Remarkably, may testing in the (synchronous/asynchronous) π -calculus [14, 16] cannot distinguish bound names from free ones; thus, a bound name can be replaced with any name in a trace. This is *not* the case here: indeed, bound names

can be always considered as addresses of nodes, while free names cannot. This makes a difference for an external observer; thus, a law like

$$\varrho \cdot \langle l'' \rangle @ l \cdot (\varrho' [l''/l']) \leq_0 \varrho \cdot (\nu l') \langle l' \rangle @ l \cdot \varrho'$$

(that, mutatis mutandis, holds for the asynchronous π -calculus [16]) does not hold for μKLAIM .

5.3.1 Soundness w.r.t. May Testing

To prove that trace equivalence exactly captures may testing we rely on the classical definition of the latter equivalence [65], as proposed in Definition 5.1.4. By using the LTS, \xrightarrow{OK} corresponds to $\xrightarrow{\langle \text{test} \rangle @ \text{test}}$; when it is convenient, we still use OK to denote label $\langle \text{test} \rangle @ \text{test}$. We start by extending the complementation function of labels to traces as expected:

$$\bar{\varrho} = \begin{cases} \epsilon & \text{if } \varrho = \epsilon \\ \bar{\phi} \cdot \bar{\varrho}' & \text{if } \varrho = \phi \cdot \varrho' \end{cases}$$

Remarkably, $\bar{\bar{\varrho}} = \varrho$. The first Lemma describes a sufficient and a necessary condition for the success of an experiment.

Lemma 5.3.2 *Let N be a net and K be an observer. Then*

1. $N \xrightarrow{\varrho}$ and $K \xrightarrow{\bar{\varrho} \cdot OK}$ imply that $N \parallel K \xrightarrow{OK}$;
2. $N \parallel K \xrightarrow{OK}$ implies that there exists a ϱ such that $N \xrightarrow{\varrho}$ and $K \xrightarrow{\bar{\varrho} \cdot OK}$.

Proof:

1. The proof is by induction on the length of ϱ . The base step is trivial. For the inductive step, we have that $\varrho = \phi \cdot \varrho'$ and we consider the possibilities for ϕ . All the cases are trivial, except for the following two:

- $\phi = (\nu \bar{l}) t \triangleleft l$. Now, we have that $N \Rightarrow N' \xrightarrow{(\nu \bar{l}) t \triangleleft l} N'' \xrightarrow{\varrho'}$, for $\bar{l} \cap \text{fn}(N') = \emptyset$; moreover, $K \Rightarrow K' \xrightarrow{(\nu \bar{l}) \langle t \rangle @ l} K'' \xrightarrow{\bar{\varrho}' \cdot OK}$. By induction, $N'' \parallel K'' \xrightarrow{OK}$. Now, by Proposition 5.2.3(2), $K' \equiv (\nu \bar{l})(K'' \parallel l :: \langle t \rangle)$ and $K'' \equiv K''' \parallel l :: \mathbf{nil}$; thus, $N \parallel K \Rightarrow (\nu \bar{l})(N' \parallel K''' \parallel l :: \langle t \rangle) \xrightarrow{\tau} (\nu \bar{l})(N'' \parallel K'') \xrightarrow{OK}$ (indeed, since K is an observer, it can only emit test at test ; thus, $\text{test} \notin \bar{l}$).
- $\phi = (\nu \bar{l}) \langle t \rangle @ l$. This case is symmetric.

2. By definition, it must be that $N \parallel K \xrightarrow{(\tau)^n H} \xrightarrow{OK}$; the proof is by induction on n . The base step is simple: $\varrho = \epsilon$. For the inductive step, we have two sub-cases:

- $N \parallel K \xrightarrow{\tau} N' \parallel K' (\xrightarrow{\tau})^{n-1} H$. By induction, $N' \xrightarrow{\varrho'}$ and $K' \xrightarrow{\varrho' \cdot OK}$, for some ϱ' . There are six possibilities for the first τ -step:
 - (a) $N \xrightarrow{\tau} N'$ and $K' \equiv K$: in this case, we can pick up $\varrho = \varrho'$.
 - (b) $N' \equiv N$ and $K \xrightarrow{\tau} K'$: like before.
 - (c) $N \xrightarrow{\triangleright l} N'$ and $K \xrightarrow{\mathbf{nil} @ l} K'$: we can pick up $\varrho = \triangleright l \cdot \varrho'$.
 - (d) $N \xrightarrow{\mathbf{nil} @ l} N'$ and $K \xrightarrow{\triangleright l} K'$: we can pick up $\varrho = \mathbf{nil} @ l \cdot \varrho'$.
 - (e) $N \xrightarrow{t \triangleleft l} N'$ and $K \xrightarrow{\langle t \rangle @ l} K'$: we can pick up $\varrho = t \triangleleft l \cdot \varrho'$.
 - (f) $N \xrightarrow{\langle t \rangle @ l} N'$ and $K \xrightarrow{t \triangleleft l} K'$: we can pick up $\varrho = \langle t \rangle @ l \cdot \varrho'$.
- $N \parallel K \xrightarrow{\tau} (\nu \tilde{l})(N' \parallel K') (\xrightarrow{\tau})^{n-1} H$, for $\tilde{l} \neq \emptyset$. Since $H \equiv (\nu l')H'$ for some H' (indeed, τ -steps cannot remove restrictions), it must be $\text{test} \notin \tilde{l}$. Thus, $N' \parallel K' (\xrightarrow{\tau})^{n-1} H' \xrightarrow{OK}$ and, by induction, $N' \xrightarrow{\varrho'}$ and $K' \xrightarrow{\varrho' \cdot OK}$, for some ϱ' . There are only two possibilities for the first τ -step:
 - (a) $N \xrightarrow{(\nu \tilde{l}) \langle t \rangle @ l} N'$ and $K \xrightarrow{t \triangleleft l} K'$. By definition of the LTS, we have to extend the scope of \tilde{l} before passing it by using rule (STR-EXT). Thus, $\tilde{l} \cap \text{fn}(K) = \emptyset$ and, by rule (LTS-Rcv), we have that $K \xrightarrow{(\nu \tilde{l}) t \triangleleft l} K'$. Thus, we can pick up $\varrho = (\nu \tilde{l}) \langle t \rangle @ l \cdot \varrho'$.
 - (b) $N \xrightarrow{t \triangleleft l} N'$ and $K \xrightarrow{(\nu \tilde{l}) \langle t \rangle @ l} K'$: this case is symmetric. ■

The next Lemma states that the laws in Table 5.2 are ‘sound’, in the sense that, if an observer can observe a trace ϱ (i.e., that can provide $\bar{\varrho}$), then it can also observe any trace $\varrho' \leq \varrho$.

Lemma 5.3.3 *Let $\varrho' \leq \varrho$ and $N \xrightarrow{\bar{\varrho}}$; then, $N \xrightarrow{\bar{\varrho}'}$.*

Proof: By definition, $\varrho' \leq_0 \varrho$; we proceed by induction on n . The base step is trivial, by reflexivity. For the inductive step, we let $\varrho' \leq_0 \varrho'' \leq_0 \varrho$; it suffices to prove that $N \xrightarrow{\bar{\varrho}}$ implies that $N \xrightarrow{\bar{\varrho}'}$. Indeed, by induction, the latter judgement implies that $N \xrightarrow{\bar{\varrho}'}$, as required. We reason by case analysis on the law in Table 5.2 used to infer $\varrho'' \leq_0 \varrho$. Notice that all the laws hide a double formulation that is made explicit in this proof.

(L1).a: $\varrho \triangleq \varrho_1 \cdot (\nu \tilde{l}) t \triangleleft l \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot (\nu \tilde{l}) \varrho_2$. By definition, $N \xrightarrow{\bar{\varrho}_1} N' \xrightarrow{(\nu \tilde{l}) \langle t \rangle @ l} N'' \xrightarrow{\bar{\varrho}_2}$; moreover, $N' \equiv (\nu \tilde{l})(N''' \parallel l :: \langle t \rangle)$ and $N'' \equiv N''' \parallel l :: \mathbf{nil}$. Now, if $\tilde{l} = \emptyset$ or $\tilde{l} \notin \text{fn}(\varrho_2)$, then it must be that $N' \xrightarrow{\bar{\varrho}_2}$ and, hence, $N \xrightarrow{\bar{\varrho}'}$. Otherwise, we only consider the case for $\tilde{l} = \{l'\}$; the more general case, where $|\tilde{l}| > 1$, is only notationally

more complex and can be recovered with a straightforward induction. By hypothesis, $\varrho_2 \triangleq \varrho_3 \cdot (\widetilde{v\bar{l}}) t' \triangleleft l'' \cdot \varrho_4$ for $l'' \in t' - \text{fn}(\varrho_3, l'', \widetilde{l'})$; thus, $N'' \xrightarrow{\overline{\varrho_3}} N''_1 \xrightarrow{(\widetilde{v\bar{l}}) \langle t' \rangle @ l''} N''_2 \xrightarrow{\overline{\varrho_4}}$. Now, $N' \xrightarrow{\overline{\varrho_3}} (\widetilde{v\bar{l}})(N''_1 \parallel l :: \langle t \rangle \parallel l'' :: \langle t' \rangle)$ and hence $N \xrightarrow{\overline{\varrho_1 \cdot \varrho_3 \cdot (\widetilde{v\bar{l}}) \langle t' \rangle @ l'' \cdot \overline{\varrho_4}}} N \xrightarrow{\overline{\varrho''}}$, i.e. $N \xrightarrow{\overline{\varrho''}}$.

(L1).b: $\varrho \triangleq \varrho_1 \cdot \triangleright l \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot \varrho_2$. By definition, $N \xrightarrow{\overline{\varrho_1}} N' \xrightarrow{\text{nil} @ l} N'' \xrightarrow{\overline{\varrho_2}}$; moreover, $N' \equiv N''$ and hence $N \xrightarrow{\overline{\varrho_1 \cdot \varrho_2}}$, as required.

(L2).a: $\varrho \triangleq \varrho_1 \cdot (\widetilde{v\bar{l}}) t \triangleleft l \cdot \phi \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot (\widetilde{v\bar{l}})(\phi \cdot t \triangleleft l \cdot \varrho_2)$. By definition, $N \xrightarrow{\overline{\varrho_1}} N' \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} N'' \xrightarrow{\overline{\phi}} N''' \xrightarrow{\overline{\varrho_2}}$; moreover, $N' \equiv (\widetilde{v\bar{l}})(\hat{N} \parallel l :: \langle t \rangle)$ and $N'' \equiv \hat{N} \parallel l :: \text{nil}$. Now, if $\widetilde{l} = \emptyset$ or $\widetilde{l} \cap \text{fn}(\phi) = \emptyset$, then it must be that $N' \xrightarrow{\overline{\phi \cdot (\widetilde{v\bar{l}}) \langle t \rangle @ l \cdot \overline{\varrho_2}}} N''$ and, hence, $N \xrightarrow{\overline{\varrho''}}$. Otherwise, it must be $\phi = (\widetilde{v\bar{l}}) t' \triangleleft l''$; thus, $N'' \xrightarrow{(\widetilde{v\bar{l}}) \langle t' \rangle @ l''} N'''$. Now, if we let $\widetilde{l}_1 = \widetilde{l} \cap (t' - \widetilde{l'})$ and $\widetilde{l}_2 = \widetilde{l} - \widetilde{l}_1$, we have that $N' \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} N'' \xrightarrow{(\widetilde{v\bar{l}}) \langle t' \rangle @ l''} (\widetilde{v\bar{l}})(N''' \parallel l :: \langle t \rangle) \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} N'''' \parallel l :: \text{nil} \xrightarrow{\overline{\varrho_2}}$, and hence $N \xrightarrow{\overline{\varrho''}}$.

(L2).b: $\varrho \triangleq \varrho_1 \cdot \triangleright l \cdot \phi \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot \phi \cdot \triangleright l \cdot \varrho_2$. By definition, $N \xrightarrow{\overline{\varrho_1}} N' \xrightarrow{\text{nil} @ l} N'' \xrightarrow{\overline{\phi}} N''' \xrightarrow{\overline{\varrho_2}}$; moreover, $N' \equiv N'' \equiv N''' \parallel l :: \text{nil}$. This implies that $N''' \equiv N''' \parallel l :: \text{nil}$ (since nodes cannot disappear along reductions) and $N \xrightarrow{\overline{\varrho_1 \cdot \phi \cdot \text{nil} @ l \cdot \overline{\varrho_2}}} N$, as required.

(L3).a: $\varrho \triangleq \varrho_1 \cdot (\widetilde{v\bar{l}}) t \triangleleft l \cdot \langle t \rangle @ l \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot (\widetilde{v\bar{l}})\varrho_2$. By definition, $N \xrightarrow{\overline{\varrho_1}} N' \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} N''_1 \xrightarrow{t \triangleleft l} N''_2 \xrightarrow{\overline{\varrho_2}}$; moreover, $N' \equiv (\widetilde{v\bar{l}})(\hat{N} \parallel l :: \langle t \rangle)$ and $N''_1 \equiv \hat{N} \parallel l :: \text{nil}$. Thus, $N' \xrightarrow{(\widetilde{v\bar{l}})(N''_2 \parallel l :: \langle t \rangle) \xrightarrow{\tau} (\widetilde{v\bar{l}})(N'' \parallel l :: \text{nil}) \triangleq N_3$. Now, if $\widetilde{l} = \emptyset$ or $\widetilde{l} \cap \text{fn}(\varrho_2) = \emptyset$, then $N_3 \xrightarrow{\overline{\varrho_2}}$ and, hence, $N \xrightarrow{\overline{\varrho''}}$. Otherwise, we reason like in case (L1).a to obtain that $N_3 \xrightarrow{(\widetilde{v\bar{l}})\overline{\varrho_2}}$ and, again, $N \xrightarrow{\overline{\varrho''}}$.

(L3).b: $\varrho \triangleq \varrho_1 \cdot \triangleright l \cdot \text{nil} @ l \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot \varrho_2$. By definition, $N \xrightarrow{\overline{\varrho_1}} N' \xrightarrow{\text{nil} @ l} N'' \xrightarrow{\triangleright l} N''' \xrightarrow{\overline{\varrho_2}}$; moreover, $N' \equiv N'' \parallel l :: \text{nil}$. This implies that $N' \xrightarrow{\triangleright l} N'''$ and we can easily conclude.

(L4).a: $\varrho \triangleq \varrho_1 \cdot (\widetilde{v\bar{l}}) t \triangleleft l \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot \triangleright l \cdot (\widetilde{v\bar{l}}) t \triangleleft l \cdot \varrho_2$. By definition, $N \xrightarrow{\overline{\varrho_1}} N' \xrightarrow{(\widetilde{v\bar{l}}) \langle t \rangle @ l} N'' \xrightarrow{\overline{\varrho_2}}$; moreover, $N' \equiv (\widetilde{v\bar{l}})(N''' \parallel l :: \langle t \rangle)$ and $N'' \equiv N''' \parallel l :: \text{nil}$. Thus, easily, $N \xrightarrow{\overline{\varrho_1}} N' \xrightarrow{\text{nil} @ l \cdot (\widetilde{v\bar{l}}) \langle t \rangle @ l} N'' \xrightarrow{\overline{\varrho_2}}$, as required.

(L4).b: $\varrho \triangleq \varrho_1 \cdot \triangleright l \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot \triangleright l \cdot \triangleright l \cdot \varrho_2$. Similar to case (L4).a.

(L5).a: $\varrho \triangleq \varrho_1 \cdot (\nu \tilde{l}) \langle t \rangle @ l \cdot \triangleright l' \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot \triangleright l' \cdot (\nu \tilde{l}) \langle t \rangle @ l \cdot \varrho_2$, for $l' \notin \tilde{l}$. By definition, $N \xrightarrow{\bar{\varrho}_1} N_1 \xrightarrow{(\nu \tilde{l}) t \triangleleft l} N_2 \Rightarrow N_3 \xrightarrow{\mathbf{nil} @ l'} N_4 \xrightarrow{\bar{\varrho}_2}$; moreover, $N_3 \equiv N_3 \parallel l' :: \mathbf{nil} \equiv N_4$. Since $l' \notin \tilde{l}$, we have that l' must be a node also in N_2 and in N_1 , i.e., $N_2 \equiv N_2 \parallel l' :: \mathbf{nil}$ and $N_1 \equiv N_1 \parallel l' :: \mathbf{nil}$. Hence, $N \xrightarrow{\bar{\varrho}_1} N_1 \xrightarrow{\mathbf{nil} @ l' (\nu \tilde{l}) \langle t \rangle @ l} N_2 \Rightarrow N_4 \xrightarrow{\bar{\varrho}_2}$, as required.

(L5).b: $\varrho \triangleq \varrho_1 \cdot \mathbf{nil} @ l \cdot \triangleright l' \cdot \varrho_2$ and $\varrho'' \triangleq \varrho_1 \cdot \triangleright l' \cdot \mathbf{nil} @ l \cdot \varrho_2$. Similar to case (L5).a. ■

Now, the main theorem follows.

Theorem 5.3.4 (Soundness of \asymp w.r.t. \simeq) *If $N \asymp M$ then $N \simeq M$.*

Proof: Let K be an observer such that $N \parallel K \xrightarrow{OK}$. By Lemma 5.3.2.2, there exists ϱ such that $N \xrightarrow{\varrho}$ and $K \xrightarrow{\bar{\varrho} \cdot OK}$. By Definition 5.3.1, there exists $\varrho' \leq \varrho$ such that $M \xrightarrow{\varrho'}$. By suffix closure of \leq (that can be easily proved), we have that $\varrho' \cdot \text{test} \triangleleft \text{test} \leq \varrho \cdot \text{test} \triangleleft \text{test}$. By Lemma 5.3.3, $K \xrightarrow{\bar{\varrho}' \cdot OK}$. By Lemma 5.3.2.1, $M \parallel K \xrightarrow{OK}$, as required by Definition 5.1.4. Thus, $N \simeq' M$ that, by Proposition 5.1.5, implies that $N \simeq M$. ■

5.3.2 Completeness w.r.t. May Testing

To start, we define the *canonical observer* of a trace ϱ , written $\Pi(\varrho)$, as

$$\Pi(\varrho) = C[\text{test} :: P]$$

where the actual observer process P and context $C[\cdot]$ enabling the observation are returned by $O_\emptyset(\varrho) = \langle P; C[\cdot] \rangle$, which is inductively defined as follows

$$O_L(\epsilon) = \langle \mathbf{out}(\text{test}) @ \text{test} . \mathbf{nil}; [\cdot] \rangle$$

$$O_L(\mathbf{nil} @ l \cdot \varrho) = \langle \mathbf{eval}(\mathbf{nil}) @ l . P; C[\cdot] \rangle$$

where $O_L(\varrho) = \langle P; C[\cdot] \rangle$

$$O_L((\nu l_1, \dots, l_n) \langle t \rangle @ l \cdot \varrho) = \langle \mathbf{in}(t\{l_i \leftarrow !x_i\}_{i=1, \dots, n}) @ l . (P[x_1/l_1, \dots, x_n/l_n]); C[\cdot] \rangle$$

where $O_{L \cup \{l_1, \dots, l_n\}}(\varrho) = \langle P; C[\cdot] \rangle$, and
 x_1, \dots, x_n are fresh names, and
 $t\{l_i \leftarrow !x_i\}_{i=1, \dots, n}$ is the template obtained
from t by replacing
each l_i with $!x_i$

$$\begin{aligned}
O_L(\triangleright l \cdot \varrho) &= \begin{cases} \langle P; C[\cdot] \parallel l :: \mathbf{nil} \rangle & \text{if } l \notin L \\ \langle P; C[\cdot] \rangle & \text{otherwise} \end{cases} \\
&\text{where } O_L(\varrho) = \langle P; C[\cdot] \rangle \\
O_L((\widetilde{\nu}l) t \triangleleft l \cdot \varrho) &= \begin{cases} \langle \mathbf{new}(\widetilde{l}).\mathbf{out}(t)@l.P; C[\cdot] \parallel l :: \mathbf{nil} \rangle & \text{if } l \notin L \\ \langle \mathbf{new}(\widetilde{l}).\mathbf{out}(t)@l.P; C[\cdot] \rangle & \text{otherwise} \end{cases} \\
&\text{where } O_L(\varrho) = \langle P; C[\cdot] \rangle
\end{aligned}$$

L is the (finite) set of names extruded by the trace, i.e. those names created by the net that emitted $\underline{\varrho}$ and offered as a datum in a visible location. We used the convention that $\mathbf{new}(\widetilde{l}).\mathbf{out}(l')@l$ stands for $\mathbf{out}(l')@l$ whenever $\widetilde{l} = \emptyset$. The context has only to provide localities where

- the observed net can place data/code (when ϱ is of the form $\triangleright l \cdot \varrho'$)
- the observer process can place data that the observed net needs (when ϱ is of the form $(\widetilde{\nu}l) t \triangleleft l \cdot \varrho'$).

However, the context should not provide a locality l' whenever $l' \in L$. In this case, the observed net already provides l' ; indeed, if $l' \in L$, then l' has been extruded by an action $(\widetilde{\nu}l', \widetilde{l}) \langle t \rangle @ l$ in ϱ .

The key property of the canonical observer for ϱ is that it always reports success when run in parallel with a net that offers ϱ , as stated by the following Proposition.

Proposition 5.3.5 $\Pi(\varrho) \xrightarrow{\overline{\varrho}.OK}$.

Proof: The proof is by induction on $|\varrho|$ and easily follows by definition of canonical observers. ■

Now, we distinguish the label $\triangleright l$ generated by rule (LTS-OUT) from the same label generated by rule (LTS-EVAL). We shall write $\boxminus l$ the former and $\triangleright l$ the latter. This is needed for technical reasons (see the case (iv) in the proof of Lemma 5.3.7); the two labels are exactly the same. We start by adapting Lemma 5.3.2(2) in order to exclude labels of the form $\boxminus l$.

Lemma 5.3.6 *If $N \parallel \Pi(\varrho) \xrightarrow{OK}$, then there exists a ϱ' such that $N \xrightarrow{\varrho'}$, $\Pi(\varrho) \xrightarrow{\overline{\varrho}.OK}$ and ϱ' does not contain labels of the form $\boxminus l$.*

Proof: By Lemma 5.3.2(2), we know that there exists a trace ϱ'' such that $N \xrightarrow{\varrho''}$ and $\Pi(\varrho) \xrightarrow{\overline{\varrho''.OK}}$. The proof now proceeds by induction on the number of labels of the form $\boxminus l$ (for a generic l) in $\overline{\varrho''}$. The base step is trivial. For the inductive step, we have that $\overline{\varrho''} = \overline{\varrho_1} \cdot \boxminus l \cdot \overline{\varrho_2}$ such that $\overline{\varrho_1}$ does not contain labels of the form $\boxminus \dots$. We consider two cases:

- There are no *intruded* names² in $\overline{\varrho}_1$. Thus, ϱ_1 does not contain labels of the form $(\widetilde{vl}) \langle t \rangle @ l''$, for $\widetilde{l} \neq \emptyset$. Let $\Pi(\varrho) \triangleq C[\text{test} :: P] \xrightarrow{\overline{\varrho}_1} C'[\text{test} :: \mathbf{out}(t)@l.P] \xrightarrow{\boxplus^l} C'[\text{test} :: P] \parallel l :: \langle t \rangle \xrightarrow{\overline{\varrho}_2}$. By definition of canonical observers, it must be that $C'[\cdot] \equiv C'[\cdot] \parallel l :: \mathbf{nil}$; indeed, for every action **out** at l in a canonical observer, a node with address l is always provided, except when l is a name intruded by the observer (i.e., extruded by the trace), that is not the case here. Thus, $\Pi(\varrho) \xrightarrow{\overline{\varrho}_1 \cdot \overline{\varrho}_2}$. On the other side, $N \xrightarrow{\varrho_1} N' \xrightarrow{\mathbf{nil} @ l} N'' \xrightarrow{\varrho_2}$, where $N' \equiv N''$; thus, $N \xrightarrow{\varrho_1 \varrho_2}$. The thesis holds by induction on $\overline{\varrho}_1 \cdot \overline{\varrho}_2$ that has one label of the form \boxplus less than $\overline{\varrho}''$.
- There are intruded names in $\overline{\varrho}_1$ and these are $\{l_1, \dots, l_k\}$. If $l \notin \{l_1, \dots, l_k\}$, then the proof is like in the case above; otherwise, let l be l_i . Since l_i has been intruded, it must be that $\Pi(\varrho) \xrightarrow{\overline{\varrho}_3} K \xrightarrow{(\widetilde{vl}_i, \widetilde{l}) t \triangleleft l''} K' \parallel l_i :: \mathbf{nil} \xrightarrow{\overline{\varrho}_4} C[\text{test} :: \mathbf{out}(t')@l_i.P]$, where $\varrho_1 = \varrho_3 \cdot (\widetilde{vl}_i, \widetilde{l}) \langle t \rangle @ l'' \cdot \varrho_4$ and $C[\text{test} :: P] \parallel l_i :: \langle t' \rangle \xrightarrow{\overline{\varrho}_2}$. Now, $N \xrightarrow{\varrho_3} N_1 \xrightarrow{(\widetilde{vl}_i, \widetilde{l}) \langle t \rangle @ l''} N_2 \xrightarrow{\varrho_4} N_3 \xrightarrow{\mathbf{nil} @ l_i} N_4 \xrightarrow{\varrho_2}$, for some $N_2 \equiv N_2 \parallel l_i :: \mathbf{nil}$ (this is always possible because in N_1 name l_i is restricted and can be used as address of a node, by using law (STR-RNODE)). Moreover, $N_3 \equiv N_4$; thus, since the node with address l_i in $K' \parallel l_i :: \mathbf{nil}$ cannot disappear during computations, it holds that $\Pi(\varrho) \xrightarrow{\overline{\varrho}_3 \cdot (\widetilde{vl}_i, \widetilde{l}) t \triangleleft l'' \cdot \overline{\varrho}_4 \cdot \overline{\varrho}_2}$, i.e. $\Pi(\varrho) \xrightarrow{\overline{\varrho}_1 \cdot \overline{\varrho}_2}$, and correspondingly $N \xrightarrow{\varrho_1 \varrho_2}$. Like before, we can apply induction to $\overline{\varrho}_1 \cdot \overline{\varrho}_2$ and conclude. ■

The main Lemma to prove completeness of trace equivalence w.r.t. may testing is the following one, stating that $\Pi(\varrho)$ can report success only upon execution of a trace ϱ' such that $\overline{\varrho}' \leq \varrho$.

Lemma 5.3.7 *Let $\Pi(\varrho) \xrightarrow{\varrho' \cdot OK}$, where $\text{test} \notin n(\varrho')$ and ϱ' does not contain labels of the form $\boxplus l$. Then, $\overline{\varrho}' \leq \varrho$.*

Proof: The proof is by induction on $|\varrho|$. The base step is trivial. For the inductive step, let ϱ be $\phi \cdot \varrho''$; let us reason on the possibilities for ϕ .

(i) $\phi \triangleq \mathbf{nil} @ l$. By construction, $\Pi(\varrho) \triangleq C[\text{test} :: \mathbf{eval}(\mathbf{nil})@l.P]$, where $\langle P; C[\cdot] \rangle = O_\emptyset(\varrho'')$. The trace $\Pi(\varrho) \xrightarrow{\varrho' \cdot OK}$ can be produced only in two ways:

1. $\varrho' \triangleq \varrho_1 \cdot \triangleright l \cdot \varrho_2$, where $C[\mathbf{0}] \xrightarrow{\varrho_1} C'[\mathbf{0}]$ and $C'[\text{test} :: P] \xrightarrow{\varrho_2 \cdot OK}$. Thus, $\Pi(\varrho'') \triangleq C[\text{test} :: P] \xrightarrow{\varrho_1 \varrho_2 \cdot OK}$; by induction, this implies that $\overline{\varrho}_1 \cdot \overline{\varrho}_2 \leq \varrho''$. Now, $\varrho \triangleq \phi \cdot \varrho'' \geq \phi \cdot \overline{\varrho}_1 \cdot \overline{\varrho}_2 \geq \overline{\varrho}_1 \cdot \phi \cdot \overline{\varrho}_2 \triangleq \overline{\varrho}'$, where

²By symmetry of denomination w.r.t. extruded names, we call *intruded* a name received via rule (LTS-Rcv), i.e. name l' is intruded in ϱ if $\varrho = \varrho' \cdot (\widetilde{vl}', \widetilde{l}') t \triangleleft l' \cdot \varrho''$.

the first inequality holds by prefix closure of \geq (i.e., the inverse of \leq) while the second inequality holds by repeated applications of law (L5). Indeed, since $C[\cdot]$ is just a parallel of nodes with no components, ϱ_1 only contains labels of the form **nil** @ $_$; thus, $\overline{\varrho_1}$ only contains labels of the form $\triangleright _$.

$$2. \varrho' \triangleq \varrho_1 \cdot \varrho_2, \text{ where } C[\mathbf{0}] \xrightarrow{\varrho_1 \cdot \mathbf{nil} @ l} C'[\mathbf{0}] \text{ and } C'[\text{test} :: P] \xrightarrow{\varrho_2 \cdot OK} .$$

Thus, $\Pi(\varrho'') \triangleq C[\text{test} :: P] \xrightarrow{\varrho_1 \cdot \mathbf{nil} @ l \cdot \varrho_2 \cdot OK}$; by induction, this implies that $\overline{\varrho_1} \cdot \triangleright l \cdot \overline{\varrho_2} \leq \varrho''$. Now, by prefix closure, by repeated applications of law (L5) (like in the previous case) and by law (L3), we have that $\varrho \triangleq \phi \cdot \varrho'' \geq \phi \cdot \overline{\varrho_1} \cdot \triangleright l \cdot \overline{\varrho_2} \geq \overline{\varrho_1} \cdot \triangleright l \cdot \phi \cdot \overline{\varrho_2} \geq \overline{\varrho_1} \cdot \overline{\varrho_2} \triangleq \overline{\varrho'}$, as required.

(ii) $\phi \triangleq \triangleright l$. By construction, $\Pi(\varrho) \triangleq C[\text{test} :: P] \parallel l :: \mathbf{nil}$, where $\langle P; C[\cdot] \rangle = O_\emptyset(\varrho'')$. Now, we have that $\varrho' \triangleq \varrho_1 \cdot \varrho_2$, where $C[\text{test} :: P] \xrightarrow{\varrho_1} C'[\text{test} :: P']$ and $C'[\text{test} :: P'] \parallel l :: \mathbf{nil} \xrightarrow{\varrho_2 \cdot OK}$. Now, $\Pi(\varrho'') \triangleq C[\text{test} :: P] \xrightarrow{\varrho_1 \cdot \varrho_2' \cdot OK}$, where ϱ_2' is the trace obtained from ϱ_2 by removing all the labels **nil** @ l from it and by possibly adding a label of the form $\triangleright l$. Indeed, since it is not necessarily the case that $C[\cdot] \equiv \dots \parallel l :: \mathbf{nil}$, it can be that some labels **nil** @ l cannot be generated by $C'[\text{test} :: P']$; similarly, it could be necessary to add a label $\triangleright l$ if, in the production of ϱ_2 , $C'[\text{test} :: P']$ needs l to place some data/process. Hence, by induction, we have that $\overline{\varrho_1} \cdot \overline{\varrho_2'} \leq \varrho''$. We now have the desired $\varrho \geq \phi \cdot \overline{\varrho_1} \cdot \overline{\varrho_2'} \geq \phi \cdot \overline{\varrho_1} \cdot \overline{\varrho_2} \geq \overline{\varrho_1} \cdot \overline{\varrho_2}$. Notice that the second inequality has been obtained by repeated applications of laws (L4) and (L2) (as many times as the number of labels **nil** @ l removed from ϱ_2 to obtain ϱ_2') and by possibly applying laws (L4), (L2) and (L3) (if a label of the form $\triangleright l$ has been introduced in ϱ_2'). The last inequality relies on law (L1).

(iii) $\phi \triangleq (\nu l_1, \dots, l_n) \langle t \rangle @ l$. By construction, $\Pi(\varrho) \triangleq C[\text{test} :: \mathbf{in}(t\{l_i \leftarrow !x_i\}_{i=1, \dots, n}) @ l.(P^{x_1/l_1, \dots, x_n/l_n})]$, where $\langle P; C[\cdot] \rangle = O_{\{l_1, \dots, l_n\}}(\varrho'')$. Now, we have that $\varrho' \triangleq \varrho_1 \cdot (\nu l_1, \dots, l_n) t \triangleleft l \cdot \varrho_2$, where $\Pi(\varrho) \xrightarrow{\varrho_1} C'[\text{test} :: \mathbf{in}(t\{l_i \leftarrow !x_i\}_{i=1, \dots, n}) @ l.(P^{x_1/l_1, \dots, x_n/l_n})] \xrightarrow{(\nu l_1, \dots, l_n) t \triangleleft l} C'[\text{test} :: P] \parallel \prod_{i=1, \dots, n} l_i :: \mathbf{nil} \xrightarrow{\varrho_2 \cdot OK}$. Let us now pick up any $l' \in \{l_1, \dots, l_n\}$; we say that l' occurs in *target position* in ϱ'' if ϱ'' contains labels of the form $\triangleright l'$ or $_ \triangleleft l'$. By an easy inspection of the definition of canonical observers, it holds that $\Pi(\varrho'')$ is structurally equivalent to $C[\text{test} :: P] \parallel l' :: \mathbf{nil}$, if l' occurs in target position in ϱ'' , or to $C[\text{test} :: P]$, otherwise.

We now proceed like in case (ii) above and let $\Pi(\varrho'') \xrightarrow{\varrho_1 \cdot \varrho_2' \cdot OK}$ where ϱ_2' is obtained from ϱ_2 by removing actions **nil** @ l' and by possibly adding an action $\triangleright l'$, for each l' occurring in target position in ϱ'' . The proof is then

similar, but uses (L5) to place ϕ at its right place and needs to be iterated for each l' occurring in target position in ϱ'' .

(iv) $\phi \triangleq (\nu \tilde{l}) t \triangleleft l$. By construction, $\Pi(\varrho) \triangleq C[\text{test} :: \mathbf{new}(\tilde{l}).\mathbf{out}(t)@l.P] \parallel l :: \mathbf{nil}$, where $\langle P; C[\cdot] \rangle = O_\emptyset(\varrho'')$. This case is the most tedious: $\Pi(\varrho)$ has a lot of possible evolutions and, thus, ϱ' can be of several forms. However, by hypothesis ϱ' does not contain labels of the form $\boxtimes _$; in particular, it does not contain $\boxtimes l$. Hence, the action $\mathbf{out}(t)@l$ does not generate any visible action and forces $\Pi(\varrho)$ to reduce to $(\nu \tilde{l})(C[\text{test} :: P] \parallel l :: \langle t \rangle)$ in order to report success. We have to keep into account whether and how $C[\text{test} :: P]$ and $l :: \langle t \rangle$ interact, and whether and how \tilde{l} are extruded. We have 6 possibilities in total.

1. $l :: \langle t \rangle$ is not involved in the generation of ϱ' and $\tilde{l}' \subseteq \tilde{l}$ are extruded by $C[\text{test} :: P]$.

Remark: here and in the rest of this proof we shall always let \tilde{l}' be a single name l' . The case for $\tilde{l}' = \emptyset$ is simpler, while the case for $|\tilde{l}'| > 1$ is only notationally more complex.

In this case, we have that $\varrho' \triangleq \varrho_1 \cdot (\nu l', \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_2$ and $\Pi(\varrho'') \xrightarrow{\varrho_1 \cdot (\nu \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_2 \cdot OK}$. By using induction and prefix closure, we have that $\varrho \geq \phi \cdot \overline{\varrho_1} \cdot (\nu \tilde{l}'') t' \triangleleft l'' \cdot \overline{\varrho_2} \geq \overline{\varrho_1} \cdot (\nu l', \tilde{l}'') t' \triangleleft l'' \cdot \overline{\varrho_2} \triangleq \overline{\varrho'}$, where the second inequality relies on law (L1) and the names in $\tilde{l} - \{l'\}$ disappear, since they are not extruded by ϱ' (and, thus, they do not occur therein).

2. the first contribution of $l :: \langle t \rangle$ in ϱ' is with label $\mathbf{nil} @ l$ and $\tilde{l}' \subseteq \tilde{l}$ are extruded. We have three sub-cases.

(a) the datum $\langle t \rangle$ is not used: in this case, $\varrho' \triangleq \varrho_1 \cdot (\nu l', \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_2 \cdot \mathbf{nil} @ l \cdot \varrho_3$ and $\Pi(\varrho'') \xrightarrow{\varrho_1 \cdot (\nu \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_2 \cdot \varrho_3 \cdot OK}$, where ϱ_3' has been obtained from ϱ_3 like in (ii) before.

By induction and prefix closure, we can prove that $\varrho \geq \phi \cdot \overline{\varrho_1} \cdot (\nu \tilde{l}'') t' \triangleleft l'' \cdot \overline{\varrho_2} \cdot \overline{\varrho_3'} \geq \triangleright l \cdot \phi \cdot \overline{\varrho_1} \cdot (\nu \tilde{l}'') t' \triangleleft l'' \cdot \overline{\varrho_2} \cdot \overline{\varrho_3'} \geq \overline{\varrho_1} \cdot (\nu l', \tilde{l}'') t' \triangleleft l'' \cdot \overline{\varrho_2} \cdot \triangleright l \cdot \overline{\varrho_3} \triangleq \overline{\varrho'}$, where the second inequality holds by law (L4), the third inequality holds by law (L1) and because l' does not appear in what follows and the fourth one is obtained by using (L2), (L3) and (L4) like in case (ii) above.

Remark: it could also be $\varrho' \triangleq \varrho_1 \cdot \mathbf{nil} @ l \cdot \varrho_2 \cdot (\nu l', \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_3$. This case can be easily adapted, by considering $\Pi(\varrho'') \xrightarrow{\varrho_1 \cdot \varrho_2 \cdot (\nu \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_3 \cdot OK}$, where ϱ_i' has been obtained from ϱ_i like in (ii).

- (b) it then offers the datum via a label $(\nu \tilde{l}_1) \langle t \rangle @ l$, for $\tilde{l}_1 = \tilde{l} - \tilde{l}'$ and \tilde{l}' have been previously extruded: in this case, $\varrho' \triangleq \varrho_1 \cdot$

$\mathbf{nil} @ l \cdot \varrho_2 \cdot (v\tilde{l}_2) \langle t' \rangle @ l'' \cdot \varrho_3 \cdot (v\tilde{l}_1) \langle t \rangle @ l \cdot \varrho_4$ and $\Pi(\varrho'') \xrightarrow{\varrho_1 \cdot \varrho'_2 \cdot (v\tilde{l}_2) \langle t' \rangle @ l'' \cdot \varrho'_3 \cdot \varrho'_4 \cdot OK}$, where the ϱ'_i 's have been obtained from the corresponding ϱ_i 's like in (ii) above. The proof proceeds like in the previous cases, by using (L4), (L2) and (L3). Moreover, like in case 2.(a), the proof is not radically changed if we consider $\varrho' \triangleq \varrho_1 \cdot (v\tilde{l}_2) \langle t' \rangle @ l'' \cdot \varrho_2 \cdot \mathbf{nil} @ l \cdot \varrho_3 \cdot (v\tilde{l}_1) \langle t \rangle @ l \cdot \varrho_4$.

- (c) *the datum $\langle t \rangle$ is then passed to $C[\text{test} :: P]$ with a communication and $\tilde{l}' \subseteq \tilde{l}$ are extruded:* in this case, $\varrho' \triangleq \varrho_1 \cdot \mathbf{nil} @ l \cdot \varrho_2 \cdot (v\tilde{l}', \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_3 \cdot \varrho_4$ and $\Pi(\varrho'') \xrightarrow{\varrho_1 \cdot \varrho'_2 \cdot (v\tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho'_3 \cdot t \triangleleft l \cdot \varrho'_4 \cdot OK}$, where the ϱ'_i 's have been obtained from the corresponding ϱ_i 's like in (ii) above. Then, $\varrho \geq \triangleright l \cdot \phi \cdot \overline{\varrho}_1 \cdot \overline{\varrho}_2 \cdot (v\tilde{l}'') \langle t' \rangle @ l'' \cdot \overline{\varrho}_4 \cdot \langle t \rangle @ l \cdot \overline{\varrho}_3 \geq \overline{\varrho}_1 \cdot \triangleright l \cdot \overline{\varrho}_2 \cdot (v\tilde{l}'') \langle t' \rangle @ l'' \cdot \phi \cdot \langle t \rangle @ l \cdot \overline{\varrho}_4 \geq \overline{\varrho}_1 \cdot \triangleright l \cdot \overline{\varrho}_2 \cdot \overline{\varrho}_3 \cdot \overline{\varrho}_4 \triangleq \overline{\varrho}'$, where the first step relies on induction, prefix closure and law (L4), the second step on laws (L2), (L4) and (L3) used as needed, and the third one on rule (L3). The situation in which the extrusion of l' proceeds the label $\mathbf{nil} @ l$ is similar.

Remark: *if l' is extruded after the communication, ϱ_3 will be of the form $\dots \langle l' \rangle @ l'' \dots$; hence, $\overline{\varrho}_3$ is $\dots \langle l'' \rangle \dots$. Now, $(v\tilde{l}')\overline{\varrho}_3 = \dots \langle v\tilde{l}' \rangle \triangleleft l'' \dots$ and the proof carries on in the same way.*

3. *the first contribution of $l :: \langle t \rangle$ in ϱ' is with $(v\tilde{l}_1) \langle t \rangle @ l$, where $\tilde{l}_1 = \tilde{l} - \tilde{l}'$ and \tilde{l}' have been previously extruded:* in this case, $\varrho' \triangleq \varrho_1 \cdot (v\tilde{l}', \tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_2 \cdot (v\tilde{l}_1) \langle t \rangle @ l \cdot \varrho_3$ and $\Pi(\varrho'') \xrightarrow{\varrho_1 \cdot (v\tilde{l}'') \langle t' \rangle @ l'' \cdot \varrho_2 \cdot \varrho_3 \cdot OK}$, where ϱ'_3 has been obtained from ϱ_3 like in (ii). The thesis follows by induction, prefix closure, law (L2) and by possibly repeated applications of laws (L4), (L2) and (L3).
4. *the first contribution of $l :: \langle t \rangle$ to the production of ϱ' is by passing the datum to $C[\text{test} :: P]$ with a communication and $\tilde{l}' \subseteq \tilde{l}$ are extruded:* the proof of this case can be adapted from case 2.(c) above. ■

Finally, we can prove that trace equivalence is a sound proof technique for may testing (see Theorem 5.3.4) that exactly captures it.

Theorem 5.3.8 (Completeness of \approx w.r.t. \simeq) *If $N \simeq M$ then $N \approx M$.*

Proof: Let ϱ be a trace of N , i.e. $N \xrightarrow{\varrho}$. By Proposition 5.3.5, $\Pi(\varrho) \xrightarrow{\overline{\varrho} \cdot OK}$; thus, by Lemma 5.3.2.1, $N \parallel \Pi(\varrho) \xrightarrow{OK}$. By Proposition 5.1.5 and Definition 5.1.4, it holds that $M \parallel \Pi(\varrho) \xrightarrow{OK}$. By Lemma 5.3.6, there exists ϱ' such that $M \xrightarrow{\varrho'}$, $\Pi(\varrho) \xrightarrow{\overline{\varrho'} \cdot OK}$ and ϱ' does not contain labels of the form $\boxtimes l$. By Lemma 5.3.7

(notice that, since `test` is fresh, it holds that `test` $\notin n(\varrho')$), $\varrho' \leq \varrho$ as required by Definition 5.3.1; thus, $N \asymp M$. ■

Corollary 5.3.9 (Tractable Characterisation of May Testing) $\asymp = \simeq$.

5.4 Verifying a Protocol for the Dining Philosophers

We now use the proof techniques we have just presented to state and prove the properties of a ‘classical’ problem in distributed systems, namely the ‘*Dining Philosophers*’.³ In what follows, we shall use bisimulation, that is finer but easier to prove. All the work can be done with trace equivalence as well.

The problem. The dining philosophers is a “classical” synchronisation problem; its luck derives from the fact that it naturally models many synchronisation problems arising when allocating resources in concurrent/distributed systems. The problem can be described as follows. Some, say n , philosophers spend their lives alternating between thinking and eating. They are seated around a circular table and there is a fork placed between each pair of neighbouring philosophers. Each philosopher has access to the forks at his left and right; if a philosopher wants to eat, he has to acquire both the forks near to him (this is possible only if none of his neighbours are using the forks); when done eating, the philosopher puts both forks back down on the table and begins thinking. The challenge in the dining philosophers problem is to design a protocol so that the philosophers do not deadlock (i.e. the entire set of philosophers does not stop and wait indefinitely), and so that no philosopher starves (i.e. each philosopher eventually gets his hands on a pair of forks). Additionally, the protocol should be as efficient as possible – in other words, the time that philosophers spend waiting to eat should be minimised.

Our solution. We now propose a protocol in μKLAIM to solve the problem, in the same spirit as Dijkstra’s solution. We shall associate each philosopher with a distinct locality taken from the set $\{l_1, \dots, l_n\}$. We also use a restricted locality l to record in a tuple of length n the allocation of the forks (i.e. the status of each philosopher); more precisely, if the i -th component of this tuple is `t` then the i -th philosopher is thinking, if it is `e` the i -th philosopher is eating. The access to such a tuple will allow the processes to act on resources allocation in mutual exclusion. The node l_i will then host the following process (implementing the behaviour of the i -th philosopher):

$$\mathbf{rec} X. \mathbf{think} . \mathbf{in}(T_i)@l. \mathbf{out}(t_i)@l. \mathbf{eat} . \mathbf{in}(T'_i)@l. \mathbf{out}(t'_i)@l.X$$

³Historically, the problem was first formulated and solved by Dijkstra in 1965 and was used to motivate the use of semaphores.

where

$$\begin{aligned}
T_i &\triangleq \begin{cases} \mathbf{t}, \mathbf{t}, !x_3, \dots, !x_{n-1}, \mathbf{t} & \text{if } i = 1 \\ !x_1, \dots, !x_{i-2}, \mathbf{t}, \mathbf{t}, \mathbf{t}, !x_{i+2}, \dots, !x_n & \text{if } 1 < i < n \\ \mathbf{t}, !x_2, \dots, !x_{n-2}, \mathbf{t}, \mathbf{t} & \text{if } i = n \end{cases} \\
t_i &\triangleq \begin{cases} \mathbf{e}, \mathbf{t}, x_3, \dots, x_{n-1}, \mathbf{t} & \text{if } i = 1 \\ x_1, \dots, x_{i-2}, \mathbf{t}, \mathbf{e}, \mathbf{t}, x_{i+2}, \dots, x_n & \text{if } 1 < i < n \\ \mathbf{t}, x_2, \dots, x_{n-2}, \mathbf{t}, \mathbf{e} & \text{if } i = n \end{cases} \\
T'_i &\triangleq \begin{cases} \mathbf{e}, \mathbf{t}, !y_3, \dots, !y_{n-1}, \mathbf{t} & \text{if } i = 1 \\ !y_1, \dots, !y_{i-2}, \mathbf{t}, \mathbf{e}, \mathbf{t}, !y_{i+2}, \dots, !y_n & \text{if } 1 < i < n \\ \mathbf{t}, !y_2, \dots, !y_{n-2}, \mathbf{t}, \mathbf{e} & \text{if } i = n \end{cases} \\
t'_i &\triangleq \begin{cases} \mathbf{t}, \mathbf{t}, y_3, \dots, y_{n-1}, \mathbf{t} & \text{if } i = 1 \\ y_1, \dots, y_{i-2}, \mathbf{t}, \mathbf{t}, \mathbf{t}, y_{i+2}, \dots, y_n & \text{if } 1 < i < n \\ \mathbf{t}, y_2, \dots, y_{n-2}, \mathbf{t}, \mathbf{t} & \text{if } i = n \end{cases}
\end{aligned}$$

Intuitively, the first **in** action verifies that the neighbours of the i -th philosopher are not eating and simultaneously acquires the lock on the tuple; then the **out** action sets the status of the i -th philosopher to **e**, while releasing the lock. Then, the following **in** and **out** actions release the resources used upon completion of the eating phase and the protocol iterates.

For the sake of simplicity, we do not model the *think* phase, while the *eat* phase is just an **out** action over some (fresh) locality l' . Hence, if the system starts with all the philosophers in a thinking state, the net implementing the system is

$$\begin{aligned}
N &\triangleq (\nu l)(l :: \langle \mathbf{t}, \dots, \mathbf{t} \rangle \parallel \prod_{i=1}^n l_i :: P_i) \\
P_i &\triangleq \mathbf{rec } X.\mathbf{in}(T_i)@l.\mathbf{out}(t_i)@l.\mathbf{out}(l_i)@l'.\mathbf{in}(T'_i)@l.\mathbf{out}(t'_i)@l.X
\end{aligned}$$

Soundness of our solution. We shall now verify the correctness of our protocol, namely that (1) no deadlock nor starvation ever occur, (2) resources are properly used (namely, no neighbouring philosophers eat at the same time) and (3) the protocol enables the highest level of parallelism (i.e. it is possible for $\lfloor \frac{n}{2} \rfloor$ philosophers to eat together).

1. We shall prove that

$$N \parallel l' :: \mathbf{nil} \approx N \parallel l' :: \langle l_i \rangle \quad (5.1)$$

for each $i = 1, \dots, n$. Equation (5.1) can be proved by showing that the relations $\mathfrak{R}_1^i \triangleq \{(N', N' \parallel l' :: \langle l_i \rangle) : N \parallel l' :: \mathbf{nil} \Rightarrow N'\} \cup Id$ are weak bisimulations (up-to \equiv); this can be done easily. This means that computations from N can never get stuck (hence deadlock will never occur) and that each philosopher can eat an unbounded number of times (hence starvation cannot occur).

Deadlock freedom: To prove it, we proceed by contradiction; hence, let us suppose that there exists a computation from N leading to deadlock.

Since the computation is finite, we can find an integer k which is an upper bound to the number of steps performed by N before reaching the deadlock. But then, we can iterate $k + 1$ times Equation (5.1) and Theorem 5.2.7 to obtain that $N \parallel l' :: \mathbf{nil} \approx N \parallel \prod_{j=1}^{k+1} l' :: \langle l_i \rangle$. This equivalence is however contradicted by letting $N \parallel l' :: \mathbf{nil}$ to follow the computation leading to deadlock. Indeed, since $N \parallel l' :: \mathbf{nil}$ performs at most k steps in such computation, it is impossible for it to produce $k + 1$ data in l' (recall that l' is fresh for N); on the other hand, no computation from $N \parallel \prod_{j=1}^{k+1} l' :: \langle l_i \rangle$ will ever remove data from l' (because l' has been chosen fresh for N). Thus, the resulting nets exhibit different data in l' and cannot be equivalent.

Starvation freedom: The proof is similar. Indeed, if there exists a computation from N starving philosopher i by letting him to eat at most k times, then such a computation contradicts $N \parallel l' :: \mathbf{nil} \approx N \parallel \prod_{j=1}^{k+1} l' :: \langle l_i \rangle$.

2. Let l'' be a fresh locality. We define

$$\begin{aligned} M &\triangleq l :: \langle \mathbf{t}, \dots, \mathbf{t} \rangle \parallel \prod_{i=1}^n l_i :: P_i \\ C[\cdot] &\triangleq l'' :: \mathbf{nil} \parallel (\nu l, l')[\cdot] \\ \mathcal{D}[\cdot] &\triangleq l'' :: \mathbf{nil} \parallel (\nu l, l')(\cdot) \parallel l :: \begin{array}{l} \mathbf{in}(e, e, !x_3, \dots, !x_n)@l.\mathbf{out}()@l'' \\ | \mathbf{in}(!x_1, e, e, !x_4, \dots, !x_n)@l.\mathbf{out}()@l'' \\ | \dots \\ | \mathbf{in}(e, !x_2, \dots, !x_{n-1}, e)@l.\mathbf{out}()@l'' \end{array} \end{aligned}$$

Notice that $N \triangleq (\nu l)M$ and hence $C[M] \triangleq l'' :: \mathbf{nil} \parallel (\nu l')N$. We have restricted node l' because we are not interested in observing who is eating (and because this simplifies the formulation of Equation (5.2) below); we later show that this fact implies that N must access resources properly. We want to prove that

$$C[M] \approx \mathcal{D}[M] \quad (5.2)$$

i.e. $\mathcal{D}[M]$ will never produce data at l'' (since $C[M]$ cannot). Intuitively, $\mathcal{D}[M]$ can produce a datum at l'' if it happens that two adjacent philosophers eat simultaneously; hence Equation (5.2) implies that no resource is ever misused. The above equation can be proved by showing that the relation $\mathfrak{R}_2 \triangleq \{(C[M'], \mathcal{D}[M']) : C[M] \Rightarrow C[M']\}$ is a bisimulation; again, this is an easy task.

Now, suppose that there exists a computation from N misusing the resources; this means that $N \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_k} N'$ and N' is a net where two adjacent philosophers are eating simultaneously. Thus $N' \triangleq (\nu l)(l :: \langle \dots, e, e, \dots \rangle \parallel \dots)$ where the two e are adjacent modulo n . But then $\mathcal{D}[M] \xrightarrow{\alpha'_1} \dots \xrightarrow{\alpha'_k} \mathcal{D}[M']$ where $N' \triangleq (\nu l)M'$, $\alpha'_i = \alpha_i$ if l' is not the target of

α_i , and $\alpha'_i = \tau$ otherwise. Hence, $\mathcal{D}[M'] \xrightarrow{\tau} \xrightarrow{\tau} \langle \rangle @ l''$, thus contradicting Equation (5.2).

3. The easiest way to prove that $\lfloor \frac{n}{2} \rfloor$ philosophers can eat simultaneously is to show a computation from N leading to a tuple in l with exactly $\lfloor \frac{n}{2} \rfloor$ items of kind e , while respecting the correct use of resources. The wanted reduction is obtained by letting the even philosophers accessing in turn the status tuple. This is always possible since an even philosopher is always surrounded (modulo n) by two, not eating, odd philosophers. Hence we have that

$$\begin{aligned}
N &\xrightarrow{\tau} \xrightarrow{\tau} l :: \langle t, e, t, t, \dots, t \rangle \parallel \prod_{i=1, \dots, n}^{i \neq 2} l_i :: P_i \parallel \\
&\quad l_2 :: \mathbf{out}(l_2) @ l'. \mathbf{in}(T'_2) @ l. \mathbf{out}(t'_2) @ l. P_2 \\
&\xrightarrow{\tau} \xrightarrow{\tau} l :: \langle t, e, t, e, t, t, \dots, t \rangle \parallel \prod_{i=1, \dots, n}^{i \neq 2, 4} l_i :: P_i \parallel \\
&\quad l_2 :: \mathbf{out}(l_2) @ l'. \mathbf{in}(T'_2) @ l. \mathbf{out}(t'_2) @ l. P_2 \parallel \\
&\quad l_4 :: \mathbf{out}(l_4) @ l'. \mathbf{in}(T'_4) @ l. \mathbf{out}(t'_4) @ l. P_4 \\
&\quad \dots \\
&\xrightarrow{\tau} \xrightarrow{\tau} l :: \langle t, e, t, e, \dots \rangle \parallel \prod_{i=1, \dots, n}^{i \text{ odd}} l_i :: P_i \parallel \\
&\quad \prod_{i=1, \dots, n}^{i \text{ even}} l_i :: \mathbf{out}(l_i) @ l'. \mathbf{in}(T'_i) @ l. \mathbf{out}(t'_i) @ l. P_i
\end{aligned}$$

5.5 Equational Laws and the Impact of Richer Contexts

In this section, we want to discuss some equational laws that can be easily proved by exploiting both bisimulation and trace equivalence. We concentrate on bisimulation that is finer (by virtue of Proposition 5.1.5 and Theorems 5.2.8 and 5.3.8). The first law is inspired by the asynchronous π -calculus [5]

$$l' :: \mathbf{rec} X. \mathbf{in}(!x) @ l. \mathbf{out}(x) @ l. X \approx l' :: \mathbf{nil}$$

and states that (repeatedly) accessing a datum and putting it back in its original location is observationally equivalent to performing no operation. Of course, this heavily exploits the fact that communication in μKLAIM is asynchronous.

We have also the following four significant laws (the last one can be easily derived from the second and the third one):

$$l :: \mathbf{out}(t) @ l'. P \parallel l' :: \mathbf{nil} \approx l :: P \parallel l' :: \langle t \rangle \quad (5.3)$$

$$l :: \mathbf{eval}(Q) @ l'. P \parallel l' :: \mathbf{nil} \approx l :: P \parallel l' :: Q \quad (5.4)$$

$$l :: P | Q \parallel l' :: \mathbf{nil} \approx l :: P \parallel l' :: Q \quad (5.5)$$

$$l :: \mathbf{eval}(Q) @ l'. P \parallel l' :: \mathbf{nil} \approx l :: P | Q \parallel l' :: \mathbf{nil} \quad (5.6)$$

Laws 5.3 and 5.4 state that it is impossible to know when data and processes have been allocated – either at the outset or during computations. Law 5.5 states that,

once the net is fixed, the actual distribution of processes is irrelevant, while law 5.6 states that remotely executing a process is observationally equivalent to executing the process locally. At a first sight, these laws could be quite surprising and seem to contradict the design principles at the basis of μKLAIM . However, they can be explained by observing the net at a very high level, namely at the level of the user applications. Indeed, we are observing the functionalities a net offers to a terminal user. Therefore, the allocation of processes cannot be observed (law 5.5) and the advantages of exploiting mobile processes (e.g. efficiency, reduced network load, support for disconnected operations) cannot be perceived at all (law 5.6).

In many circumstances this level of abstraction is exactly what we need. For example, when we studied the ‘Dining philosophers’, we were interested in the overall behaviour of the system and in the properties it enjoyed; thus, we could ignore the implementation details and take into account only the functional aspects of the protocol. If we want more details on the distributed environment underlying a μKLAIM application, we have to refine the observational level. Consequently, to study lower-level aspects like, e.g., routing problems or failures, we have to adapt the language and the semantic theories we developed in this chapter. To this aim, we have studied three variants of μKLAIM where (i) communication can only take place locally, (ii) failures (of both components and nodes) can occur, and (iii) dynamically evolving connections between nodes are explicitly modelled. Later on, we shall give some hints on the first two variants and leave the more elaborated treatment of the third scenario for [63]. Predictably, laws 5.5 and 5.6 do not hold in these lower-level settings.

Local Communications. In the setting of LCKLAIM the above rules do not hold. Indeed, by letting \approx_l be may testing in LCKLAIM , it is very easy to check that

$$\begin{aligned} l :: P \parallel l' :: \mathbf{nil} &\not\approx_l l :: \mathbf{nil} \parallel l' :: P \\ l :: \mathbf{eval}(Q)@l'.P \parallel l' :: \mathbf{nil} &\not\approx_l l :: P|Q \parallel l' :: \mathbf{nil} \end{aligned}$$

This is reasonable because, since communications are local, by moving a process we also change its execution environment. Thus, at the very least, its observable behaviour will change according to the node where it runs. Notice that, in order to disprove laws 5.5 and 5.6 we have used may testing; because of Proposition 5.1.5 (that holds also for LCKLAIM), $\not\approx_l$ implies $\not\approx_l$.

Failures. Now, we consider another setting and enrich μKLAIM with a mechanism for modelling various forms of failures. This is achieved by adding the following rules to the definition of the reduction relation and of the LTS:

$$\begin{aligned} (\text{RED-FAIL}) \quad l :: C &\mapsto \mathbf{0} & (\text{LTS-FAIL}) \quad l :: C &\xrightarrow{\tau} \mathbf{0} \end{aligned}$$

These rules model corruption of data (*message omission*) if $C \triangleq \langle t_1 \rangle | \dots | \langle t_n \rangle$, node (*fail-silent*) failure if $l :: C$ collects all the components located at l , and abnormal

termination of some processes running at l otherwise. In this way, we model failures as disappearance of a resource (a datum, a process or a whole node). This is a simple, but realistic, way of representing failures, specifically fail-silent and message omission, in a global computing scenario [34]. Indeed, while the presence of data/nodes can be ascertained, their absence cannot because in such a scenario there is no practical upper bound to communication delays. Thus, failures cannot be distinguished from long delays and should be modelled as totally asynchronous and undetectable events.

Again, it is easy to prove that laws 5.5 and 5.6 given for μKLAIM do not hold anymore in this more concrete setting. Indeed, the failure of l' can easily modify the overall behaviour of the equated nets. We now examine what happens to the characterisation of barbed congruence and may testing in this new framework. The definition of the bisimulation equivalence does not need to be modified to exactly capture barbed congruence. Indeed, the recursive closure of both barbed congruence and bisimulation already forces the corruption of the same data and the failure of the same nodes to take place at the same time; as regards process abnormal termination, it will be the evolution of the involved nets that will affect the equivalence. About trace equivalence, the characterisation breaks down: trace equivalence is only a sound (but not complete) proof technique for may testing. The problem is that Lemmas 5.3.6 and 5.3.7 do not hold anymore in the lower-level setting. This does not mean that trace equivalence is strictly finer than may testing, even if we believe this; it only means that the proof of Theorem 5.3.8 must be carefully re-examined. A more precise statement on this aspect is left for future work.

If we let \approx_f , \cong_f , \succ_f and \simeq_f denote labelled bisimilarity, barbed congruence, trace equivalence and may testing for the calculus with failures, we have

Theorem 5.5.1 $\approx_f = \cong_f$ and $\succ_f \subseteq \simeq_f$.

Proof: The proof is formally identical to those of Theorems 5.2.8, 5.2.12 and 5.3.4, but now a τ -step or a reduction can be also generated by applying rule (LTS-FAIL) and (RED-FAIL) respectively. ■

5.6 Related Work

We conclude by reviewing related work on observational equivalences for calculi with process distribution and mobility (many of them are surveyed in [46]). In the nineties, many CCS-like process calculi have been enriched with localities to explicitly describe the distribution of processes. The aim was mainly to provide these calculi with non interleaving semantics or, at least, to differentiate processes' parallel components (thus obtaining more inspective semantics than the interleaving ones). This line of research is far from the one in which μKLAIM falls, where localities are used as a mean to make processes network aware thus enabling them

to refer to the network locations as target of remote communication or as destination of migrations. Localities are not only considered as units of distribution but, according to the case, as units of mobility, of communication, of failure or of security.

[130] and [4] extend, resp., CCS and π -calculus with process distribution and mobility. In both cases, processes run over the nodes of an explicit, flat and dynamically evolving net architecture. Nodes can fail thus causing loss of all hosted processes. There are explicit operations to kill nodes and to query the status of a node. Failures can be detected, which is suitable for distributed computing but clashes with the assumptions underlying global computing. In both papers, a labelled bisimulation (akin to the bisimulation in the CCS and π -calculus) is given to capture a standardly defined barbed congruence.

Another distributed version of the π -calculus is presented in [85]; the resulting calculus contains primitives for code movement and creation of new localities/channels in a net with a flat architecture. Over the LTS defining the semantics of the calculus, a typed bisimulation (with a tractable formulation) is defined that exactly capture typed barbed equivalence. The use of types illustrates the importance of having the rights to observe a given behaviour: indeed, different typings (i.e. observation rights) generate different bisimulations, that are finer as long as the typing is less restrictive. Clearly, typed equivalences can be also introduced in the framework of μ KLAIM. This would result in a very powerful and interesting formalism, considering the impact that the types presented in Chapter 3 have in the semantics of the calculus. We leave this aspect for a future work; nevertheless, we strongly conjecture that this could be done without too many problems, by merging together the theory presented in this chapter with the ideas put forward in [85].

In the Distributed Join calculus [71], located mobile processes are hierarchically structured and form a tree-like structure evolving during the computation. Entire subtrees, not only single processes, can move). Technically, nets are flat collections of named nodes, where the name of a node indicates the nesting path of the node; e.g., a node whose name is $l_1. \dots .l_k.l$ represents a node referable to via the unique name l and that is contained in l_k , that is a node contained in l_{k-1} and so on. Communication in DJoin takes place in two steps: firstly, the sending process sends a message on a channel; then, the ether (i.e. the environment containing all the nodes) delivers the message to the (unique) process that can receive on that channel. The fact that in the whole net there is a unique process capable to receive at a given channel makes DJoin communication somehow similar to μ KLAIM one, in that DJoin channels have a role similar to that of μ KLAIM localities. Failures are modelled by tagging locality names: e.g. the (compound) name $\dots .l_i^\Omega. \dots .l$ states that l is a node contained in a failed node l_i and, thus, l itself is failed. The Ω at l_i has been caused by execution of the primitive *halt* by a process running at l_i . Failures can be detected by using the primitive *fail*. Failed nodes cannot host running computations but can receive data/code/sublocations that, however, once arrived in the failed node, become definitely stuck. Some interesting laws and properties are proved using a contextual barbed equivalence, but no tractable characterisation of

the equivalence is given and it is not even obvious how to extend the characterisation of barbed bisimulation for the (non-distributed) Join calculus introduced in [72] to account for distribution and agent mobility.

The Ambient calculus [41] is an elegant notation to model hierarchically structured distributed applications. Though the definition of its reduction semantics is very simple, the formulation of a reasonable, possibly tractable, observational equivalence is a very hard task. The calculus is centred around the notion of connections between ambients, that are containers of processes and data. Each primitive can be executed only if the ambient hierarchy is structured in a precise way; e.g., an ambient n can enter an ambient m only if n and m are sibling, i.e. they are both contained in the same ambient. This fact greatly complicates the definition of a tractable equivalence. Recently, in [108], a bisimulation capturing Ambient's barbed congruence has been defined. This has been done by structuring the syntax into two levels, namely processes and nets (where the latter ones are particular cases of the former ones), and by exploiting an involved LTS (using three different kinds of labels some of which containing process contexts). However, the defined bisimulation is not standard and suffers from a quantification over all the possible processes (to fill in the 'holes' generated by the operational semantics).

Similar bisimulations have also been developed for calculi derived from Ambient, like, e.g., Safe Ambients [107], Boxed Ambients [28], the Seal Calculus [44] and the calculus of Mobile Resources [77]. Moreover, in the last three papers, bisimulation is only a sound but not complete proof technique for barbed congruence.

To conclude, we want to remark that, to the best of our knowledge, no characterisation of may testing in terms of trace equivalence has even been given for an asynchronous, distributed language with process mobility. In [142], a theory for may testing (and the corresponding characterisation) is developed for the Actors Model [3]. However, the work is done by reducing Actors to a typed asynchronous π -calculus and the trace-based characterisation follows [16].

Chapter 6

Expressiveness of the Languages

To conclude this thesis, we now try to assess the expressive power of tuple based communications and to evaluate the theoretical impact of the linguistic primitives proposed for the language KLAIM. This task is performed by studying the possibility of encoding each of the calculi presented in Chapter 2 in a more basilar one. A tight comparison between these calculi and the asynchronous π -calculus [93, 19], named π_a -calculus, is also provided. At the end of this chapter, we shall use these results to justify the use of μ KLAIM throughout the previous chapters of the thesis. For the moment, we just say that it represents a good compromise between the expressive power of KLAIM and that of the π_a -calculus.

To assess the quality of our encodings, we shall use some well-established criteria (see, e.g., [119]), namely *full abstraction* and *semantical equivalence*. To this aim, we assume a family of equivalences EQ .

Full Abstraction w.r.t. EQ : An encoding $enc(\cdot)$ of language \mathcal{X} into language \mathcal{Y} satisfies this property if for every pair of \mathcal{X} -terms T_1 and T_2 it holds that $T_1 EQ_{\mathcal{X}} T_2$ if and only if $enc(T_1) EQ_{\mathcal{Y}} enc(T_2)$.

Semantical Equivalence w.r.t. EQ : An encoding $enc(\cdot)$ of language \mathcal{X} into language \mathcal{Y} satisfies this property if for every \mathcal{X} -term T it holds that $T EQ_{\mathcal{Z}} enc(T)$, for some language \mathcal{Z} containing both \mathcal{X} and \mathcal{Y} .

In the above definitions, EQ is not a precise equivalence but a *family* of equivalences in that it has to be properly instantiated to the various languages considered, yielding $EQ_{\mathcal{X}}$, $EQ_{\mathcal{Y}}$ and $EQ_{\mathcal{Z}}$. The stronger the equivalence the better the encoding, in that it more strongly attests that the target language has similar expressive power to the source one. Moreover, we have that if an encoding is semantical equivalent w.r.t. EQ then it is also fully abstract w.r.t. the same equivalence. Thus, an encoding enjoying semantical equivalence is ‘better’ than an encoding enjoying fully abstraction. Finally, if we want to establish semantical equivalence between a language \mathcal{Y} that is a sub-language of \mathcal{X} , then the natural choice for \mathcal{Z} is \mathcal{X} itself.

The equivalences we use in this chapter are *barbed bisimilarity* and *barbed congruence*; these are uniformly defined equivalences on process calculi often considered their ‘touchstone’ semantic theories. Barbed bisimilarity equates two terms that offer the same observable behaviour along all possible computations, while barbed congruence is obtained by closing the former under all possible language contexts. As usual, see e.g. [138], barbed bisimilarity is coarser than barbed congruence. It often turns out that a ‘half-way’ solution between the two notions above is the appropriate one; it relies on what we call *translated barbed congruence*, written \cong^{tr} . We say that an encoding $\text{enc}(\cdot)$ from language \mathcal{X} to language \mathcal{Y} is fully abstract w.r.t. \cong^{tr} whenever the set of contexts in \mathcal{Y} considered for context closure is formed by using only the translation via $\text{enc}(\cdot)$ of contexts in \mathcal{X} . Indeed, if we consider the encoding as a protocol (i.e. a precise sequence of message exchanges), translated contexts represent opponents conforming to the protocol. To assess the expressiveness of languages, this result suffices since it precisely says that the source language can be faithfully compiled in the target one.

6.1 Technical Preliminaries

In this section, we list all the technical tools we need to carry on the proofs in this chapter. We start by presenting all the machineries for μKLAIM ; the corresponding notions for cKLAIM and LcKLAIM are derived easily and are postponed to the end of this section.

As we already said, we shall assess the quality of our encodings by using a notion of *translated barbed congruence*. Once fixed an encoding $\text{enc}(\cdot)$ from a certain language \mathcal{Z} into μKLAIM , this equivalence is defined like barbed congruence but it only considers those contexts that are the encoding (via enc) of a source one. By following [13], we shall denote this barbed congruence as $\cong_{\mu K}^{\text{tr}}$ (because the contexts considered are always *translated*, via enc). However, in the proofs, it will be convenient to keep track of the number of τ -steps that a net requires to simulate the other while establishing barbed congruence. This gives rise to a preorder on nets that we call *barbed expansion*. Recall from Notation 5.2.1 that $N \xrightarrow{\hat{\tau}} N'$ stands for either $N \equiv N'$ or $N \xrightarrow{\tau} N'$.

Definition 6.1.1 (Barbed Expansion Preorder) *A preorder \mathfrak{R} between μKLAIM nets is a barbed expansion if, for each $N_1 \mathfrak{R} N_2$, it holds that:*

1. if $N_1 \downarrow l$ then $N_2 \Downarrow l$;
2. if $N_2 \downarrow l$ then $N_1 \Downarrow l$;
3. if $N_1 \xrightarrow{\tau} N'_1$ then $N_2 \xrightarrow{\tau} N'_2$ and $N'_1 \mathfrak{R} N'_2$, for some N'_2 ;
4. if $N_2 \xrightarrow{\tau} N'_2$ then $N_1 \xrightarrow{\hat{\tau}} N'_1$ and $N'_1 \mathfrak{R} N'_2$, for some N'_1 ;
5. $C[N_1] \mathfrak{R} C[N_2]$, for every context $C[\cdot]$.

The expansion preorder, $\lesssim_{\mu K}$, is the largest barbed expansion (when notationally useful, we write $N \lesssim_{\mu K} M$ as $M \gtrsim_{\mu K} N$).

Like barbed congruence, barbed expansion can be defined by requiring closure only under a subset of language contexts. In particular, once fixed an encoding $enc(\cdot)$ from a certain language \mathcal{Z} into μKLAIM , we define $\lesssim_{\mu K}^{\text{tr}}$, the *translated* barbed expansion, to be the largest relation defined like $\lesssim_{\mu K}$, but where context closure only consider those contexts $C[\cdot]$ such that $C[\cdot] = enc(\mathcal{D}[\cdot])$ and $\mathcal{D}[\cdot]$ is a \mathcal{Z} -context. We let $enc(\mathcal{D}[\cdot])$ be defined as a standard net encoding that replaces $[\cdot]$ with $[\cdot]$. We now establish an ordering among the relations introduced so far.

Proposition 6.1.2 $\equiv \subset \lesssim_{\mu K} \subset \cong_{\mu K}$ and $\equiv \subset \lesssim_{\mu K}^{\text{tr}} \subset \cong_{\mu K}^{\text{tr}}$.

Proof: We just prove the first statement; the second one is similar. The inclusion $\equiv \subset \lesssim_{\mu K}$ is simple: proving ‘ \subseteq ’ is straightforward, while the first four statements of Proposition 6.1.6 can be used to prove that the reverse inclusion does not hold. The inclusion $\lesssim_{\mu K} \subset \cong_{\mu K}$ holds by definition. ■

In what follows, we shall use some well-established proof techniques, namely *up-to expansion techniques*. We say that \mathfrak{R} is a barbed congruence up-to $\lesssim_{\mu K}$ if it is defined like in Definition 5.1.3 but reduction and context closure are weakened and consider $\gtrsim_{\mu K} \mathfrak{R} \lesssim_{\mu K}$ (instead of \mathfrak{R}) in the closure. The translated versions of barbed congruence and expansion are modified similarly. Formally, we have the following definitions.

Definition 6.1.3 (Barbed Congruence up-to $\lesssim_{\mu K}$) A symmetric relation between μKLAIM nets \mathfrak{R} is a barbed congruence up-to $\lesssim_{\mu K}$ if, whenever $N_1 \mathfrak{R} N_2$, it holds that:

- if $N_1 \downarrow l$ then $N_2 \Downarrow l$;
- if $N_1 \xrightarrow{\tau} N'_1$ then there exists N'_2 such that $N_2 \Rightarrow N'_2$ and $N'_1 \gtrsim_{\mu K} \mathfrak{R} \lesssim_{\mu K} N'_2$;
- for every context $C[\cdot]$, it holds that $C[N_1] \gtrsim_{\mu K} \mathfrak{R} \lesssim_{\mu K} C[N_2]$.

Definition 6.1.4 (Translated Barbed Congruence up-to $\lesssim_{\mu K}^{\text{tr}}$) A symmetric relation between μKLAIM nets \mathfrak{R} is a translated barbed congruence up-to $\lesssim_{\mu K}^{\text{tr}}$ if, whenever $N_1 \mathfrak{R} N_2$, it holds that:

- if $N_1 \downarrow l$ then $N_2 \Downarrow l$;
- if $N_1 \xrightarrow{\tau} N'_1$ then there exists N'_2 such that $N_2 \Rightarrow N'_2$ and $N'_1 \gtrsim_{\mu K}^{\text{tr}} \mathfrak{R} \lesssim_{\mu K}^{\text{tr}} N'_2$;
- $C[N_1] \gtrsim_{\mu K}^{\text{tr}} \mathfrak{R} \lesssim_{\mu K}^{\text{tr}} C[N_2]$, for every translated context $C[\cdot]$.

Proposition 6.1.5 (Up-to Techniques) The following facts hold:

1. if \mathfrak{R} is a barbed congruence up-to $\lesssim_{\mu K}$, then $\mathfrak{R} \subseteq \cong_{\mu K}$.
2. if \mathfrak{R} is a translated barbed congruence up-to $\lesssim_{\mu K}^{\text{tr}}$, then $\mathfrak{R} \subseteq \cong_{\mu K}^{\text{tr}}$.

Proof: The proofs of the two claims are similar; we just show the first one. It suffices to prove that $\mathfrak{S} \triangleq \{(N, M) : N \succ_{\mu K} \mathfrak{R} \lesssim_{\mu K} M\}$ is barb preserving, reduction closed and closed under translated contexts. We consider $N \succ_{\mu K} N_1 \mathfrak{R} M_1 \lesssim_{\mu K} M$. Let $N \xrightarrow{\tau} N'$. Then, by hypothesis, $N_1 \xrightarrow{\hat{\tau}} N_2$ and $N' \succ_{\mu K} N_2$. Now, if $N_1 \equiv N_2$, we can state that $N' \succ_{\mu K} N_1$; hence, $M \Rightarrow M$ and $N' \mathfrak{S} M$. On the other hand, if $N_1 \xrightarrow{\tau} N_2$ then $M_1 \xrightarrow{\hat{\tau}} M_2$ and $N_2 \succ_{\mu K} \mathfrak{R} \lesssim_{\mu K} M_2$. Then, $M \xrightarrow{\hat{\tau}} M'$ and $M_2 \lesssim_{\mu K} M'$; hence, by transitivity of $\lesssim_{\mu K}$ (that can be easily proved), we obtain $N' \mathfrak{R} M'$, as required. Now, let $N \downarrow l$; then, $N_1 \downarrow l$. Then, $M_1 \Downarrow l$, i.e. $M_1 \Rightarrow M_2 \downarrow l$. Now, $M \Rightarrow M'$ and $M_2 \lesssim_{\mu K} M'$; thus, $M' \Downarrow l$ and, hence, $M \Downarrow l$, as required. Finally, context closure holds by definition. ■

We now give some simple laws that greatly simplify our proofs.

Proposition 6.1.6 *The following facts hold:*

1. $(\nu l')(l :: P\sigma \parallel l' :: \mathbf{nil}) \lesssim_{\mu K} (\nu l')(l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle)$ whenever $\mathit{match}(T, t) = \sigma$
2. $l :: P \parallel l' :: \langle t \rangle \lesssim_{\mu K} l :: \mathbf{out}(t)@l'.P \parallel l' :: \mathbf{nil}$
3. $l :: P \parallel l' :: Q \lesssim_{\mu K} l :: \mathbf{eval}(Q)@l'.P \parallel l' :: \mathbf{nil}$
4. $(\nu l')(l :: P \parallel l' :: \mathbf{nil}) \lesssim_{\mu K} l :: \mathbf{new}(l').P$
5. $(\nu l)(l :: I) \lesssim_{\mu K} \mathbf{0} \lesssim_{\mu K} (\nu l)(l :: I)$.

Technicalities for cKLAIM and lCKLAIM. Most of the theory presented for μ KLAIM can be easily adapted to cKLAIM and lCKLAIM. In particular, an LTS for cKLAIM can be obtained from the rules in Table 3.8 by removing the rule for action **read** and by only considering monadic tuples/templates. The LTS for lCKLAIM is furtherly obtained by replacing the rules of cKLAIM for actions **out** and **in** with the following ones:

$$l :: \mathbf{out}(l').P \xrightarrow{\tau} l :: P \mid \langle l' \rangle \qquad \frac{\mathit{match}(T, l') = \sigma}{l :: \mathbf{in}(T).P \xrightarrow{l' \triangleleft l} l :: P\sigma}$$

Then, we denote with \cong_{cK} the restriction of $\cong_{\mu K}$ to cKLAIM nets; clearly, $\cong_{cK} \subseteq \cong_{\mu K}$. Relations \lesssim_{cK} , \cong_{cK}^{tr} and \cong_{cK}^{tr} are defined similarly. Finally, we define similar relations \lesssim_{lCK} and \cong_{lCK} for lCKLAIM. Clearly, all the properties stated and proved in this section for μ KLAIM can be faithfully rephrased to deal with the sub-relations containing only cKLAIM or lCKLAIM nets.

6.2 KLAIM vs μ KLAIM

Intuitions There are two differences between KLAIM and μ KLAIM: presence/absence of allocation environments and presence/absence of higher-order

communications. Intuitively, allocation environments are translated into tuples of the TS allocated at a reserved locality env . If the allocation environment ρ of l maps x to l' , then a tuple $\langle l, x, l' \rangle$ is stored at env . Hence, when performing an action **out/in/read**, all the (originally) free variables occurring in the tuple/template must be translated according to the current allocation environment. This is made possible by adding a sequence of actions **read** to properly translate the free variables. Notice, however, that a renaming of the free variables with fresh ones is necessary not to capture occurrences of the same variables within the scope of prefixed actions **eval** (this is necessary to correctly implement the dynamic binding of these variables). Informally, the KLAIM node

$$l_1 ::_{\rho_1} P$$

with

$$P \triangleq \mathbf{out}(x, l')@y.\mathbf{eval}(\mathbf{out}(x, l')@y)@x \quad (6.1)$$

and ρ_1 such that $\rho_1(x) = l_1$ and $\rho_1(y) = l_2$, is translated into the μ KLAIM net

$$l_1 :: P' \quad \parallel \quad \text{env} :: \langle l, x, l_1 \rangle \mid \langle l, y, l_2 \rangle \mid \dots$$

where

$$P' \triangleq \mathbf{read}(l, x, !x')@y.\mathbf{read}(l, y, !y')@y.\mathbf{out}(x', l')@y.\mathbf{eval}(\mathbf{out}(x, l')@y)@x' \quad (6.2)$$

Since the name binding discipline implemented for actions **out** is static, the theory developed for the higher-order π -calculus [134] by means of *triggers* can be smoothly integrated to the present setting. In *loc.cit.*, a HO π -calculus process

$$\bar{a}\langle p \rangle \quad \mid \quad a(X).X$$

is translated to

$$(vc)(\bar{a}\langle c \rangle \mid !c().\dot{p}) \quad \mid \quad a(x).\bar{x}\langle \rangle$$

where $\bar{a}\langle p \rangle$ sends process p on channel a and \dot{p} is the translation of p (for a more precise syntax and semantics of the π -calculus see Section 6.5.1). The idea of this encoding is to assign a fresh pointer c to p and distribute it in place of p . Such pointer is then used by the interested processes to activate as many copies of p as needed. This idea can be faithfully adapted to KLAIM. For example, the net

$$l_1 ::_{\rho_1} \mathbf{out}(P)@l_1 \quad \parallel \quad l_2 ::_{\rho_2} \mathbf{in}(!X)@l_1.X$$

where P is defined like in (6.1), is translated into

$$l_1 :: \mathbf{new}(l).\mathbf{eval}(P_l)@l.\mathbf{out}(l)@l_1 \quad \parallel \quad l_2 :: \mathbf{in}(!x)@l_1.\mathbf{out}(l_2)@x \quad \parallel \quad \text{env} :: \dots$$

where

$$P_l \triangleq \mathbf{rec} X.\mathbf{in}(!z)@l.(X \mid \mathbf{eval}(P')@z)$$

and P' is defined like in (6.2).

As this intuitive discussion should have clarified, name translation and handling of higher-order data are compatible issues. In particular, the full abstraction result of [134] can be established in our framework as well. Nevertheless, a formal presentation of the complete encoding turns out to be notationally overcomplicated. Thus, from now on, we only consider the first-order fragment of KLAIM , i.e. those KLAIM nets that do not contain processes in tuple fields.

Formal development We now formalise the way in which we can simulate in μKLAIM the translation via allocation environments of free variables to locality names. This is done by the encoding presented in Table 6.1, where `env` is a reserved name.

As already said, `env`'s TS collects tuples of the form $\langle l, x, l' \rangle$ to properly record the associations in l 's allocation environment. Moreover, node `env` also contains another kind of tuples, i.e. pairs $\langle l', l \rangle$ stating that the allocation environment of l' coincides with l 's one, except for the `self` entry. This is useful when l' is a node created by l . Indeed, we do not duplicate the allocation environment of l in `env` for l' , but we just put a “link” to the original environment; we shall say that l is an *alias* for l' . Clearly, this solution imposes the special handling of variable `self`, that is not implemented as the other entries of an allocation environment but is automatically resolved by the encoding (see the second case for the encoding of action **eval** and the side conditions **(**)** and **(***)** for actions **out**, **in** and **read**). Moreover, if l created l' that, in turn, created l'' , then `env` contains the tuples $\langle l', l \rangle$ and $\langle l'', l \rangle$ (see the encoding of action **new**). This is necessary because the allocation environment of l'' is, in fact, the environment of l . Thus, when performing an action **out/in/read**, the translation of the (originally) free variables must be preceded by an action **read** that retrieves the link to the proper allocation environment.

Notice that, when a locality l is present in N , its allocation environment is explicitly stored in `env` and l is clearly linked to itself (i.e., the tuple $\langle l, l \rangle$ is stored in `env`). Notice also that, by definition of $\langle \mathbf{0} \rangle$, the tuple space of `env` is never empty. This will turn out to be fundamental in order to obtain a fully abstraction result. Moreover, notice that structurally equivalent nets (like $\mathbf{0}$ and $\mathbf{0} \parallel \mathbf{0}$) may have different encodings. Nevertheless, this is not a problem, since we work with translated barbed congruence, that ignores this fact.

The main encoding relies on an auxiliary encoding for node components. Then, the component C located in l is encoded as $\langle \langle C \rangle \rangle_{l, fv(C)}$. This encoding uses `env` for operations related to environments, keeps track of the locality where the component is located (to statically resolve occurrences of variable `self` and to dynamically enable the encoded term to properly translate the free variables occurring in actions **out/in/read**) and records the originally free variables occurring in C . This last information is necessary because the encoding proceeds compositionally; thus, it is necessary to distinguish which variables were free ‘at the beginning’ from those that are temporarily free but will be bound by a binding prefix during the encoding

Encoding Nets (where <code>env</code> is a reserved name):	
$\langle\langle \mathbf{0} \rangle\rangle$	$\triangleq \text{env} :: \langle \rangle$
$\langle\langle N_1 \parallel N_2 \rangle\rangle$	$\triangleq \langle\langle N_1 \rangle\rangle \parallel \langle\langle N_2 \rangle\rangle$
$\langle\langle (\nu l)N \rangle\rangle$	$\triangleq (\nu l)\langle\langle N \rangle\rangle$
$\langle\langle l ::_{\rho} C \rangle\rangle$	$\triangleq l :: \langle\langle C \rangle\rangle_{t:fv(C)} \parallel \text{env} :: \langle l, l \rangle \mid \prod_{\substack{x \neq \text{self} \\ (x, l') \in \rho}} \langle l, l' \rangle$
Encoding Components:	
$\langle\langle t \rangle\rangle_{u;V}$	$\triangleq \langle t \rangle$
$\langle\langle C_1 C_2 \rangle\rangle_{u;V}$	$\triangleq \langle\langle C_1 \rangle\rangle_{u;V} \mid \langle\langle C_2 \rangle\rangle_{u;V}$
$\langle\langle \mathbf{nil} \rangle\rangle_{u;V}$	$\triangleq \mathbf{nil}$
$\langle\langle X \rangle\rangle_{u;V}$	$\triangleq X$
$\langle\langle \mathbf{rec} X.P \rangle\rangle_{u;V}$	$\triangleq \mathbf{rec} X.\langle\langle P \rangle\rangle_{u;V}$
$\langle\langle \mathbf{new}(l).P \rangle\rangle_{u;V}$	$\triangleq \mathbf{new}(l).\mathbf{read}(u, !y)@ \text{env}.\mathbf{out}(l, y)@ \text{env}.\langle\langle P \rangle\rangle_{u;V}$ y is fresh
$\langle\langle \mathbf{eval}(Q)@v.P \rangle\rangle_{u;V}$	$\triangleq \begin{cases} \mathbf{eval}(\langle\langle Q \rangle\rangle_{v;V})@v.\langle\langle P \rangle\rangle_{u;V} & \text{if } v \in \mathcal{L} \\ \mathbf{eval}(\langle\langle Q \rangle\rangle_{u;V})@u.\langle\langle P \rangle\rangle_{u;V} & \text{if } v = \mathbf{self} \\ \mathbf{read}(u, !y)@ \text{env}.\mathbf{read}(y, v, !z)@ \text{env}.\mathbf{eval}(\langle\langle Q \rangle\rangle_{z;V})@z.\langle\langle P \rangle\rangle_{u;V} & \text{if } (*) \end{cases}$
$\langle\langle \mathbf{out}(t)@v.P \rangle\rangle_{u;V}$	$\triangleq \mathbf{read}(u, !y)@ \text{env}.\mathbf{read}(y, x_1, !y_1)@ \text{env}.\dots$ where (**) $\dots.\mathbf{read}(y, x_n, !y_n)@ \text{env}.\mathbf{out}(t')@v'.\langle\langle P \rangle\rangle_{u;V}$
$\langle\langle \mathbf{in}(T)@v.P \rangle\rangle_{u;V}$	$\triangleq \mathbf{read}(u, !y)@ \text{env}.\mathbf{read}(y, x_1, !y_1)@ \text{env}.\dots$ where (***) $\dots.\mathbf{read}(y, x_n, !y_n)@ \text{env}.\mathbf{in}(T')@v'.\langle\langle P \rangle\rangle_{u;V}$
$\langle\langle \mathbf{read}(T)@v.P \rangle\rangle_{u;V}$	$\triangleq \mathbf{read}(u, !y)@ \text{env}.\mathbf{read}(y, x_1, !y_1)@ \text{env}.\dots$ where (***) $\dots.\mathbf{read}(y, x_n, !y_n)@ \text{env}.\mathbf{read}(T')@v'.\langle\langle P \rangle\rangle_{u;V}$
(*) $v \in V - \{\mathbf{self}\}$ and y, z are fresh	
(**) $\{x_1, \dots, x_n\} = (fv(t, v) - \{\mathbf{self}\}) \cap V$ and y, y_1, \dots, y_n are fresh and $t' = t[u, y_1, \dots, y_n/\mathbf{self}, x_1, \dots, x_n]$ and $v' = v[u, y_1, \dots, y_n/\mathbf{self}, x_1, \dots, x_n]$	
(***) $\{x_1, \dots, x_n\} = (fv(T, v) - \{\mathbf{self}\}) \cap V$ and y, y_1, \dots, y_n are fresh and $T' = T[u, y_1, \dots, y_n/\mathbf{self}, x_1, \dots, x_n]$ and $v' = v[u, y_1, \dots, y_n/\mathbf{self}, x_1, \dots, x_n]$	

Table 6.1: Encoding KLAIM into μ KLAIM

phase. To clarify this point, consider the following process

$$P \triangleq \mathbf{in}(!x_1)@l.\mathbf{out}(x_1, x_2)@l$$

located at l' . In this process, only x_2 is (originally) free. But to encode P , we need to first encode the (sub)process $\mathbf{out}(x_1, x_2)@l$ that has two free variables: x_1 and x_2 . Hence, if we encode such a process as

$$\mathbf{read}(l', !y)@env.\mathbf{read}(y, x_1, !y_1)@env.\mathbf{read}(y, x_2, !y_2)@env.\mathbf{out}(y_1, y_2)@l$$

we would change the overall behaviour. Indeed, the binding of the first argument of action \mathbf{out} to the argument of action \mathbf{in} (programmed in P) would be lost. The right solution is

$$\mathbf{read}(l', !y)@env.\mathbf{read}(y, x_2, !y_2)@env.\mathbf{out}(x_1, y_2)@l$$

that, once prefixed by (the encoding of) action $\mathbf{in}(!x_1)@l$, properly binds variable x_1 .

To prove properties of this encoding, we first introduce a notion of *normal form* of an encoding $\langle\langle N \rangle\rangle$, written $\ll\langle\langle N \rangle\rangle$. Essentially, the normal form of an encoding is the net resulting from the execution of (what we can call) *administrative* τ -steps. Informally, these are the τ -steps introduced by the encoding and that do not correspond to any τ -step in the source net. Normal forms enjoy the desirable property of being *prompt*, i.e. any top-level action they intend to perform corresponds to an analogous action in the source term. This fact will greatly simplify our proofs.

Intuitively, $\ll\langle\langle N \rangle\rangle$ is obtained from $\langle\langle N \rangle\rangle$ by firing as many top-level ‘administrative’ actions \mathbf{read} (introduced to implement allocation environments) as possible. For example, if ρ is the allocation environment of l and the side condition $(***)$ of Table 6.1 holds, we let

$$\ll\langle\langle \mathbf{read}(T)@v.P \rangle\rangle_{l;V} \triangleq \mathbf{read}(l', x_k, !y_k)@env. \cdots \mathbf{read}(l', x_n, !y_n)@env. \mathbf{read}(T')@v'.\langle\langle P \rangle\rangle_{l;V}$$

where l' is the alias for l , $\{x_1, \dots, x_{k-1}\} \subseteq \text{dom}(\rho)$ and $x_k \notin \text{dom}(\rho)$. The idea underlying this normalisation is that, if $\ll\langle\langle \mathbf{read}(T)@v.P \rangle\rangle_{l;V}$ has a top-level action of the form $\mathbf{read}(\cdot, x, !y)@env$, then there exists a variable in $(\text{fv}(T, v) - \{\text{self}\}) \cap V$ that cannot be resolved in ρ ; thus, the original action $\mathbf{read}(T)@v$ gets stuck. Hence, as expected, also its encoding gets stuck when it tries to resolve variable x .

The above definition can be made more formal; however, for the sake of simplicity, we think that this intuitive presentation suffices. Just notice that the normalisation procedure behaves similarly when the translated action is a $\mathbf{in/out/eval}$, and it extends homomorphically to complex processes and nets. The following result states that the reduction to normal forms is performed while respecting $\approx_{\mu K}^{\text{tr}}$.

Lemma 6.2.1 $\langle\langle N \rangle\rangle \approx_{\mu K}^{\text{tr}} \ll\langle\langle N \rangle\rangle$.

Proof: To prove the thesis, we need to show that

$$\mathfrak{R} \triangleq \{ (C[H], C[K]) : \langle\langle N \rangle\rangle \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* H \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* K \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* \langle\langle N \rangle\rangle \wedge C[\cdot] \text{ is a context translated via } \langle\langle \cdot \rangle\rangle \}$$

is contained in $\approx_{\mu K}^{\text{tr}}$. Let us pick up a pair $(C[H], C[K]) \in \mathfrak{R}$ and prove that it satisfies the requirements of the definition of $\approx_{\mu K}^{\text{tr}}$.

Let $C[H] \xrightarrow{\chi} \bar{H}$ and let us reason by case analysis on χ .

$\chi = \mathbf{nil} @ l$. In this case, $\bar{H} \equiv C[H]$; moreover, since H and K have the same addresses, it trivially holds that $C[K] \xrightarrow{\chi} C[K]$ and the thesis follows up-to \equiv .

$\chi = (\nu \bar{l}) \langle t \rangle @ l$. If the datum is provided by the context, then the thesis is easy to prove. Otherwise, suppose that $H \xrightarrow{(\nu \bar{l}') \langle t \rangle @ l} H'$ and let \bar{l} be obtained from \bar{l}' by adding some names \bar{l}'' bound by $C[\cdot]$. Then, by definition of the encoding and of relation \mathfrak{R} , it must be that $N \xrightarrow{(\nu \bar{l}') \langle t \rangle @ l} N'$ and that $\langle\langle N' \rangle\rangle \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* H' \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* K' \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* \langle\langle N' \rangle\rangle$, where $K \xrightarrow{(\nu \bar{l}') \langle t \rangle @ l} K'$. In conclusion, $C[H] \equiv (\nu \bar{l}'') C_1[H] \xrightarrow{\chi} C_1[H']$, where $C_1[\cdot]$ is still a translated context; moreover, $C[K] \xrightarrow{\chi} C_1[K']$ and $C_1[H'] \mathfrak{R} C_1[K']$, as required.

$\chi = \tau$. According to Lemma 5.2.6, we have six possible sub-cases, that we now examine separately.

1. $H \xrightarrow{\tau} H'$ and $\bar{H} \equiv C[H']$. There are two possibilities for this τ -step: it can be either generated by an action **read** over **env** or not.
 - (a) In the first case, by construction, it can be that K has been obtained from H by firing also such an action **read**; hence, $C[K] \xrightarrow{\epsilon} C[K]$ and $C[H'] \mathfrak{R} C[K]$. Otherwise, K can mimic this τ -step and reduce to a K' such that $\langle\langle N \rangle\rangle \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* H' \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* K' \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* \langle\langle N \rangle\rangle$ and the thesis follows.
 - (b) On the other hand, if the τ -step of H did not involved any exchange over **env**, it must be that K can perform the same action. Indeed, actions not involving **env** can only increase while passing from H to K (no action over a locality different from **env** is touched and some new action over a locality different from **env** could be enabled by the removal of some prefixing **read** over **env**). Thus, $K \xrightarrow{\tau} K'$ such that $H' \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* K'$. We can conclude, once we prove that there exists a KLAIM net M such that $\langle\langle M \rangle\rangle \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* H'$ and $K' \left(\xrightarrow{\cdot \triangleleft \text{env}} \right)^* \langle\langle M \rangle\rangle$. But this is not difficult: if H performs a τ -step without involving **env**, this means that N (that exists by

definition of \mathfrak{K}) can perform a top-level τ -step over l , see the definition of the encoding in Table 6.1. Then, the M we were looking for is the τ -reduct of N obtained from firing the action whose encoding has been fired by H .

2. $C[\cdot] \xrightarrow{\tau} C'[\cdot]$ and $\bar{H} \equiv C'[H]$. This case is trivial.
3. $H \xrightarrow{\triangleright l} H'$, $C[\cdot] \equiv C[\cdot \parallel l :: \mathbf{nil}]$ and $\bar{H} \equiv C[H']$. Clearly, $l \neq \mathbf{env}$, otherwise H could have performed the τ -step without the contribution of the context. By definition of the normalisation and of the relation \mathfrak{K} , K has as many sending actions as H (possibly, it has some more sending action resulting from the removal of some prefix **read**); thus, $K \xrightarrow{\triangleright l} K'$ such that $H'(\xrightarrow{\cdot \triangleleft \mathbf{env}})^* K'$ and hence $C[K] \xrightarrow{\tau} C[K']$. Like in case 1.(b) above, we can find a net M such that $\langle\langle M \rangle\rangle(\xrightarrow{\cdot \triangleleft \mathbf{env}})^* H'$ and $K'(\xrightarrow{\cdot \triangleleft \mathbf{env}})^* \langle\langle M \rangle\rangle$: indeed, it is the $\triangleright l$ -reduct of N obtained from firing the action whose encoding has been fired by H .
4. $H \xrightarrow{\mathbf{nil} @ l} H'$, $C[\cdot] \equiv C'[\cdot \parallel L]$, $L \xrightarrow{\triangleright l} L'$ and $\bar{H} \equiv C'[H \parallel L']$. This case is simpler.
5. $H \xrightarrow{t \triangleleft l} H'$, $C[\cdot] \equiv C'[\cdot \parallel l :: \langle t \rangle]$ and $\bar{H} \equiv C'[H']$. Again, $l \neq \mathbf{env}$, otherwise H could have performed the τ -step without the contribution of the context. The proof is like in case 3. above.
6. $H \xrightarrow{(\widetilde{v}l) \langle t \rangle @ l} H'$, $C[\cdot] \equiv C'[\cdot \parallel L]$, $L \xrightarrow{t \triangleleft l} L'$ and $\bar{H} \equiv C'[(\widetilde{v}l)(H' \parallel L')]$. Like before.

The converse, i.e. that each χ -move of $C[K]$ can be properly replied to by $C[H]$, can be proved similarly. To prove closeness under translated contexts, let $\mathcal{D}[\cdot]$ be a translated context; we have to prove that $\mathcal{D}[C[H]] \mathfrak{K} \mathcal{D}[C[K]]$, but this holds by definition of \mathfrak{K} , once we consider the context $\mathcal{D}[C[\cdot]]$ that is still a translated context. ■

Now, we can consider the operational correspondence. Through this proof, we shall write ENV_l^ρ to indicate the tuples allocated at \mathbf{env} to implement the allocation environment ρ of node l , i.e. $\langle l, l \rangle \mid \prod_{\substack{x \neq \mathbf{self} \\ (x, l') \in \rho}} \langle l, x, l' \rangle$. To better understand the following proofs, notice that translated contexts comply with the expected interaction protocol with \mathbf{env} . In particular, they cannot count how many times a given datum appears in \mathbf{env} and cannot tell $\mathbf{env} :: ENV_l^\rho \mid \langle l', l \rangle$ and $\mathbf{env} :: ENV_l^\rho \mid ENV_{l'}^\rho$ apart.

Lemma 6.2.2 (Operational Correspondence) *Let N be a KLAIM net. Then*

1. $N \mapsto N'$ implies that $\langle\langle N \rangle\rangle \mapsto^* \succ_{\mu K}^{\text{tr}} \langle\langle N' \rangle\rangle$
2. $\langle\langle N \rangle\rangle \mapsto N'$ implies that $N \mapsto N''$ and $N' \succ_{\mu K}^{\text{tr}} \langle\langle N'' \rangle\rangle$

Proof:

1. The proof is by induction on the length of the inference for $N \mapsto N'$. For the base case, we just consider two representative cases, i.e. when N evolves by exploiting rules (RED-IN) and (RED-NEW); the other ones are similar or easier.

In the first case, we have that $N \triangleq l ::_{\rho} \mathbf{in}(T)@u.P \parallel l' ::_{\rho'} \langle t \rangle$, and we let $V = fv(\mathbf{in}(T)@u.P)$. By hypothesis, $\rho(u) = l'$ and $\mathcal{E}[\![T]\!]_{\rho}$ is defined and yields T' ; thus, $fv(T, u) \subseteq dom(\rho)$. By construction, we have that

$$\langle\langle N \rangle\rangle \triangleq l :: \mathbf{in}(T')@l'.\langle\langle P \rangle\rangle_{l;V} \parallel l' :: \langle t \rangle \parallel \mathbf{env} :: ENV_l^{\rho} \mid ENV_{l'}^{\rho'}$$

Moreover, we also know that $match(T', t) = \sigma$. By using Lemma 6.2.1, we can conclude that $\langle\langle N \rangle\rangle \mapsto \langle\langle l ::_{\rho} P\sigma \parallel l' ::_{\rho'} \mathbf{nil} \rangle\rangle \triangleq \langle\langle N' \rangle\rangle \gtrsim_{\mu K}^{\text{tr}} \langle\langle N' \rangle\rangle$, as required.

When N evolves exploiting rule (RED-NEW), then $N \triangleq l ::_{\rho} \mathbf{new}(l').P$ and $N' \triangleq (\nu l')(l ::_{\rho} P \parallel l' ::_{\rho[l'/\text{self}]} \mathbf{nil})$. It is easy to show that

$$\begin{aligned} \langle\langle N \rangle\rangle &\mapsto^* (\nu l')(l :: \langle\langle P \rangle\rangle_{l;fv(P)} \parallel l' :: \mathbf{nil} \parallel \mathbf{env} :: ENV_l^{\rho} \mid \langle l', l \rangle) \\ &\gtrsim_{\mu K}^{\text{tr}} (\nu l')(l :: \langle\langle P \rangle\rangle_{l;fv(P)} \parallel l' :: \mathbf{nil} \parallel \mathbf{env} :: ENV_l^{\rho} \mid ENV_{l'}^{\rho'}) \gtrsim_{\mu K}^{\text{tr}} \langle\langle N' \rangle\rangle \end{aligned}$$

We now consider the inductive step; we only discuss the case in which the last rule applied is (RED-STRUCT). In this case, $N \mapsto N'$ because $N \equiv M$, $M \mapsto M'$ and $M' \equiv N'$. It is easy to see that structurally equivalent nets have encodings related by $\lesssim_{\mu K}^{\text{tr}}$; thus, $\langle\langle N \rangle\rangle \gtrsim_{\mu K}^{\text{tr}} \langle\langle M \rangle\rangle$ and $\langle\langle M' \rangle\rangle \gtrsim_{\mu K}^{\text{tr}} \langle\langle N' \rangle\rangle$. By induction, we know that $\langle\langle M \rangle\rangle \mapsto^* M'' \gtrsim_{\mu K}^{\text{tr}} \langle\langle M' \rangle\rangle$, for some M'' . These two facts together imply that $\langle\langle N \rangle\rangle \mapsto^* \bar{N}$ for some \bar{N} such that $\bar{N} \gtrsim_{\mu K}^{\text{tr}} M''$. By transitivity of $\gtrsim_{\mu K}^{\text{tr}}$, we can conclude.

2. The proof is by induction on the length of the inference for $\langle\langle N \rangle\rangle \mapsto N'$. We only examine the base cases for (RED-IN) and (RED-NEW). The key observation is that, because of normalisation, $\langle\langle N \rangle\rangle$ can evolve via rule (RED-IN) only if

$$\langle\langle N \rangle\rangle \triangleq l :: \mathbf{in}(T')@l'.\langle\langle P \rangle\rangle_{l;V} \parallel l' :: \langle t \rangle \parallel \mathbf{env} :: ENV_l^{\rho} \mid ENV_{l'}^{\rho'}$$

where $V = fv(\mathbf{in}(T')@l'.P)$ and $match(T', t) = \sigma$; moreover, we also have that

$$N' \equiv l :: \langle\langle P\sigma \rangle\rangle_{l;V} \parallel l' :: \mathbf{nil} \parallel \mathbf{env} :: ENV_l^{\rho} \mid ENV_{l'}^{\rho'}$$

Now, it must be that $N \triangleq l ::_{\rho} \mathbf{in}(T)@u.P \parallel l' ::_{\rho'} \langle t \rangle$, where $\rho(u) = l'$ and $\mathcal{E}[\![T]\!]_{\rho} = T'$. This suffices to infer $N \mapsto l ::_{\rho} P\sigma \parallel l' ::_{\rho'} \mathbf{nil} \triangleq N''$ and $N' \equiv \langle\langle N'' \rangle\rangle$.

The case for (RED-NEW) is proved like before. Indeed, $N \triangleq l ::_{\rho} \mathbf{new}(l').P$, $\langle\langle N \rangle\rangle \triangleq \langle\langle N \rangle\rangle$ and $N' \equiv l :: \mathbf{read}(l, !y)@env.out(l', y)@env.\langle\langle P \rangle\rangle_{l;fv(P)} \parallel l' ::$

$\mathbf{nil} \parallel \mathbf{env} :: ENV_l^\rho \mid \langle l', l \rangle$. Thus, $N \mapsto (\nu l')(l ::_\rho P \parallel l' ::_{\rho[l'/\mathbf{se1f}]} \mathbf{nil}) \triangleq N''$ and

$$\begin{aligned} N' &\gtrsim_{\mu K}^{\text{tr}} (\nu l')(l :: \langle P \rangle_{l, \text{fv}(P)} \parallel l' :: \mathbf{nil} \parallel \mathbf{env} :: ENV_l^\rho \mid \langle l', l \rangle) \\ &\gtrsim_{\mu K}^{\text{tr}} (\nu l')(l :: \langle P \rangle_{l, \text{fv}(P)} \parallel l' :: \mathbf{nil} \parallel \mathbf{env} :: ENV_l^\rho \mid ENV_{l'}^{\rho'}) \triangleq \langle N'' \rangle \end{aligned}$$

that can be easily proved. ■

Theorem 6.2.3 (Full Abstraction w.r.t. Translated Barbed Congruence)

$N \cong_K M$ if and only if $\langle N \rangle \cong_{\mu K}^{\text{tr}} \langle M \rangle$.

Proof: We start with the ‘if’ part and prove that $\mathfrak{R} \triangleq \{(N, M) : \langle N \rangle \cong_{\mu K}^{\text{tr}} \langle M \rangle\}$ is barb preserving, reduction closed and contextual. Indeed, by Lemma 6.2.1, $\langle \cdot \rangle \gtrsim_{\mu K}^{\text{tr}} \langle \langle \cdot \rangle \rangle$; hence, the hypothesis $\langle N \rangle \cong_{\mu K}^{\text{tr}} \langle M \rangle$ implies that $\langle \langle N \rangle \rangle \cong_{\mu K}^{\text{tr}} \langle \langle M \rangle \rangle$, as needed.

- Let $N \downarrow l$; since the encoding and the normalisation preserve the barbs (this can be easily seen by the definitions of $\langle \cdot \rangle$ and $\langle \langle \cdot \rangle \rangle$), we have that $\langle \langle N \rangle \rangle \downarrow l$. Then, by hypothesis, $\langle \langle M \rangle \rangle \downarrow l$, i.e. $\langle \langle M \rangle \rangle \mapsto^* M' \downarrow l$. Now, by Lemmas 6.2.2.2 and 6.2.1, we have that there exists a net M'' such that $M \mapsto^* M''$ and $M' \gtrsim_{\mu K}^{\text{tr}} \langle \langle M'' \rangle \rangle$. By Definition 5.1.1 and Proposition 5.2.3.2/.3, we can conclude that $M'' \downarrow l$ and hence $M \downarrow l$.
- Let $N \mapsto N'$; by Lemma 6.2.2.1 this implies that $\langle \langle N \rangle \rangle \mapsto^* \gtrsim_{\mu K}^{\text{tr}} \langle \langle N' \rangle \rangle$. By hypothesis, we have that $\langle \langle M \rangle \rangle \mapsto^* \gtrsim_{\mu K}^{\text{tr}} M'$, for some M' such that $\langle \langle N' \rangle \rangle \cong_{\mu K}^{\text{tr}} M'$. By Lemmas 6.2.2.2 and 6.2.1, we have that there exists a net M'' such that $M \mapsto^* M''$ and $M' \gtrsim_{\mu K}^{\text{tr}} \langle \langle M'' \rangle \rangle$. Now, since $\lesssim_{\mu K}^{\text{tr}} \subseteq \cong_{\mu K}^{\text{tr}}$ (that can be easily verified) and by transitivity of $\cong_{\mu K}^{\text{tr}}$, we have that $\langle \langle N' \rangle \rangle \cong_{\mu K}^{\text{tr}} \langle \langle M'' \rangle \rangle$; thus, $N' \mathfrak{R} M''$, as required.
- Let us pick up a translated context $C[\cdot]$; this means that $C[\cdot] \triangleq \langle \mathcal{D}[\cdot] \rangle$. Now, if either $\mathcal{D}[N]$ or $\mathcal{D}[M]$ are undefined (i.e. they give rise to a ill-defined net) then we do not have to consider $\mathcal{D}[\cdot]$ for context closure of \mathfrak{R} . Otherwise, we have to prove that $\mathcal{D}[N] \mathfrak{R} \mathcal{D}[M]$ by knowing that $C[\langle \langle N \rangle \rangle] \cong_{\mu K}^{\text{tr}} C[\langle \langle M \rangle \rangle]$. By Lemma 6.2.1 (that can be easily extended to contexts) we have that $C[\cdot] \gtrsim_{\mu K}^{\text{tr}} \langle \langle \mathcal{D}[\cdot] \rangle \rangle$ and hence $C[\cdot] \gtrsim_{\mu K}^{\text{tr}} \langle \langle \mathcal{D} \rangle \rangle[\cdot]$; thus, $\langle \langle \mathcal{D} \rangle \rangle[\langle \langle N \rangle \rangle] \cong_{\mu K}^{\text{tr}} \langle \langle \mathcal{D} \rangle \rangle[\langle \langle M \rangle \rangle]$, i.e. $\langle \langle \mathcal{D}[N] \rangle \rangle \cong_{\mu K}^{\text{tr}} \langle \langle \mathcal{D}[M] \rangle \rangle$. By definition, we obtain the required $\mathcal{D}[N] \mathfrak{R} \mathcal{D}[M]$.

Conversely, we can similarly prove that $\mathfrak{R} \triangleq \{(\langle \langle N \rangle \rangle, \langle \langle M \rangle \rangle) : N \cong_K M\}$ is barb preserving, reduction closed and contextual. We omit the details, since they are an easy adaption of the above steps. The only tricky part is barb preservation when $\langle \langle N \rangle \rangle \downarrow \mathbf{env}$; however, since $\langle \langle M \rangle \rangle$ always has at least one (possibly useless) datum at \mathbf{env} , we also have that $\langle \langle M \rangle \rangle \downarrow \mathbf{env}$, as required. ■

To conclude this section, we want to stress that we need **env** not to be empty to preserve, e.g., the equivalence $\mathbf{0} \cong_K (\nu l)(l ::_{[\text{sel}f \rightarrow l]} \mathbf{nil})$. Once translated, these two nets become $\mathbf{env} :: \langle \rangle$ and $(\nu l)(l :: \mathbf{nil} \parallel \mathbf{env} :: \langle l, l \rangle)$, respectively, that are equivalent w.r.t. $\cong_{\mu K}^{\text{tr}}$ exactly because translated contexts cannot tell tuples $\langle \rangle$ and $\langle l, l \rangle$ apart when located at **env**.

6.3 μ KLAIM VS CKLAIM

As we already said, there are two main differences between μ KLAIM and cKLAIM: presence/absence of action **read** and polyadic/monadic communications. In this section we prove that action **read** and polyadic exchanges are not essential features of the paradigm; to the best of our knowledge, our work is the first result of this kind in a LINDA-based setting.

We start with the easier task: proving that **read** actions can be implemented in cKLAIM. Essentially, **read** behaves like **in** except for the fact that it does not remove the accessed datum. It is easy to prove that

$$l :: \mathbf{read}(T)@l'.P \cong_{\mu K} l :: \mathbf{in}(T)@l'.\mathbf{out}(\widehat{T})@l'.P$$

where $\widehat{l} \triangleq l$, $\widehat{x} \triangleq x$ and $\widehat{T_1, T_2} \triangleq \widehat{T_1}, \widehat{T_2}$. This implementation of action **read** can be extended to complex nets in the obvious way, i.e. structurally; it can be then easily proved that the resulting encoding enjoys semantical equivalence w.r.t. $\cong_{\mu K}$. We omit the details on this aspect to leave space to the second difference between μ KLAIM and cKLAIM, namely the use of polyadic/monadic data.

To softly introduce the reader to our encoding, first let us examine Milner's well-known encoding of polyadic into monadic communications for the *synchronous* π -calculus [112]. We have that:

$$\bar{a}\langle b, c \rangle \mid a(x, y)$$

becomes

$$(\nu n)\bar{a}\langle n \rangle.\bar{n}\langle b \rangle.\bar{n}\langle c \rangle \mid a(n).n(x).n(y)$$

with n fresh. Hence, a fresh name (n) is exchanged by exploiting a common channel (a); n is then used to pass the sequence of values. In the *asynchronous* π -calculus [93], Honda and Tokoro propose a slightly more complex encoding:

$$\bar{a}\langle b, c \rangle \mid a(x, y)$$

is rendered as

$$\begin{aligned} & (\nu n)(\bar{a}\langle n \rangle \mid n(\eta).(\bar{\eta}\langle b \rangle \mid n(n_2).\bar{\eta}\langle c \rangle)) \\ & \mid a(n).(\nu n_1, n_2)(\bar{n}\langle \eta \rangle \mid n_1(x).(\bar{n}\langle \eta \rangle \mid n_2(y))) \end{aligned}$$

The schema is similar to the one for the synchronous calculus. However, since output sequentialisation is not possible, different channels are needed to send the different values in the sequence.

Our encoding somehow refines Honda's one because it also has to consider the presence of pattern-matching. Hence, when encoding a polyadic communication (of μKLAIM) into a monadic one (of $c\text{KLAIM}$) we are faced with the problem of starting to access a tuple and, while scanning it, finding out that it does not match with the specified template. The solution is to then put back the part of the tuple retrieved and restart the process; of course, this introduces divergence in the encoding. The full encoding is given in Table 6.2.

Remark. The encoding in Table 6.2 is defined only for nets in which each tuple is located alone on a different clone of the node hosting it (thus, for example, $\langle l :: \langle t_1 \rangle \langle t_2 \rangle \rangle$ is not defined). To overcome this problem and let the encoding easy, we let $\langle N \rangle$ to be $\langle N' \rangle$ where $N' \equiv N$ but $\langle N' \rangle$ is defined. Notice that such N' can be always found for each N by only using rule (STR-CLONE), but is not unique, in general (indeed, we can also split processes located at the same locality). To overcome this fact, we can consider the N' (unique up-to rearrangements of parallel components) obtained from N by only using (STR-CLONE) to isolate located data.

The focus of the encoding is in the implementation of tuples and in the translation of actions **in/out**. A tuple $\langle t \rangle$ is translated into a (monadic) reference to a fresh locality l where a process, $R_l(t)$, sequentially produces the fields of the tuple and the length of the tuple plus one (this is used to properly implement the pattern matching mechanism). The fields are requested sequentially by the (translation of a) **in** action by using localities $1, 2, \dots, n, \dots$; this is necessary to maintain the order of the data in the tuple, since our calculus is asynchronous. Once the process $R_l(t)$ has accepted the request for the i -th field, it produces such a field together with an acknowledgement implemented via the reserved locality go .

Once the process translating an **in** action acquires the reference to (the locality hosting the process handling) a tuple, it first verifies whether the accessed tuple and the template used to retrieve it have the same number of fields. If it is the case, it sequentially asks for all the fields of the tuple. For the i -th tuple field u_i , the encoding of the input non-deterministically chooses whether accepting u_i (because it matches the i -th template parameter T_i), thus proceeding with the tuple scanning, or refusing it and re-establishing the original scenario (with the reference put back in its original location and the process handling the tuple rolled back). In the latter case, notice that the input has not been fired and hence the process implementing it recursively starts back its task. Clearly, this protocol is *not divergent free*; the intuition underlying it is illustrated in Table 6.3.

We now prove some interesting properties of $\langle \cdot \rangle$. In particular, we prove that a polyadic net N and its encoding are semantical equivalent w.r.t. barbed bisimulation (clearly, they cannot be equivalent w.r.t. any equivalence that is a congruence). We also prove that the encoding is *adequate w.r.t. barbed congruence*, but it is *not* fully abstract (at least, when considering all the possible monadic con-

Encoding Nets:	
$\langle 0 \rangle \triangleq \mathbf{0}$	
$\langle l :: C \rangle \triangleq \begin{cases} (v')l :: \langle l' \rangle \parallel l' :: R_{l'}(t) & \text{if } C \triangleq \langle t \rangle \text{ and } l' \text{ is fresh} \\ l :: \langle P \rangle & \text{if } C = P \end{cases}$	
$\langle N_1 \parallel N_2 \rangle \triangleq \langle N_1 \rangle \parallel \langle N_2 \rangle$	
$\langle (v)N \rangle \triangleq (v) \langle N \rangle$	
Encoding Processes:	
$\langle \mathbf{nil} \rangle \triangleq \mathbf{nil}$	
$\langle X \rangle \triangleq X$	
$\langle \mathbf{rec } X.P \rangle \triangleq \mathbf{rec } X.\langle P \rangle$	
$\langle P_1 P_2 \rangle \triangleq \langle P_1 \rangle \langle P_2 \rangle$	
$\langle \mathbf{new}(l).P \rangle \triangleq \mathbf{new}(l).\langle P \rangle$	
$\langle \mathbf{eval}(Q)@l.P \rangle \triangleq \mathbf{eval}(\langle Q \rangle)@l.\langle P \rangle$	
$\langle \mathbf{out}(t)@l.P \rangle \triangleq \mathbf{eval}(\mathbf{nil})@l.\mathbf{new}(l').\mathbf{out}(l')@l.\mathbf{eval}(R_{l'}(t))@l'.\langle P \rangle$	$l' \text{ fresh}$
$\langle \mathbf{in}(T)@l.P \rangle \triangleq \mathbf{rec } X.\mathbf{in}(!x)@l.Q_{l,x,X}^0(T; P)$	$x, X \text{ fresh}$
where	
$\bullet R_l(u_1, \dots, u_n) \triangleq S_l(u_1, \dots, u_n) \mid L_l^n$	
$\bullet S_l(u_1, \dots, u_n) \triangleq \prod_{i=1}^n \mathbf{in}(i)@l.\mathbf{new}(l_i).\mathbf{out}(\mathbf{go})@l.\mathbf{out}(l_i)@l.\mathbf{out}(u_i)@l_i$	$l_i \text{ fresh}$
$\bullet L_l^n \triangleq \mathbf{in}(\mathbf{len})@l.\mathbf{new}(l_{len}).\mathbf{out}(\mathbf{go})@l.\mathbf{out}(l_{len})@l.\mathbf{out}(n+1)@l_{len}$	$l_{len} \text{ fresh}$
$\bullet Q_{l,x,X}^k(T_1, \dots, T_n; P) \triangleq \begin{cases} \mathbf{out}(\mathbf{len})@x.\mathbf{in}(\mathbf{go})@x.\mathbf{in}(!x_{len})@x.(& \text{if } k = 0 \\ \mathbf{in}(n+1)@x_{len}.Q_{l,x,X}^1(T_1, \dots, T_n; P) & \\ \mathbf{in}(!y)@x_{len}.\mathbf{eval}(L_l^n)@x.\mathbf{out}(x)@l.X &) \\ \mathbf{out}(k)@x.\mathbf{in}(\mathbf{go})@x.\mathbf{in}(!x_k)@x.(& \text{if } 1 \leq k \leq n \\ \mathbf{in}(T_k)@x_k.Q_{l,x,X}^{k+1}(T_1, \dots, T_n; P) & \\ \mathbf{in}(!y)@x_k.\mathbf{eval}(L_x^n \mid S_x(\widehat{T}_1, \dots, \widehat{T}_k))@x.\mathbf{out}(x)@l.X &) \\ \langle P \rangle & \text{if } k = n + 1 \end{cases}$	
with x_{len}, y and x_k fresh variables	
$\bullet \widehat{T} \triangleq \begin{cases} u & \text{if } T = u \\ x & \text{if } T = !x \end{cases}$	
$\bullet \mathbf{len}, \mathbf{go}, 1, \dots, n, \dots$ are pairwise distinct reserved localities	

Table 6.2: Encoding The Polyadic Calculus into the Monadic Calculus

Tuple Consumer (with template T)	Tuple Handler (for tuple t)
Acquire the lock over a tuple	
Ask for t 's length	→
If $k = T $ then proceed, otherwise release the lock and roll back the tuple handler	←
	Provide t 's length k
Ask for t 's fi rst fi eld	→
If the fi rst fi eld of T matches f then proceed, otherwise release the lock and roll back the tuple handler	←
	Provide t 's fi rst fi eld f and an ack go
...	...
Ask for t 's last fi eld	→
If the last fi eld of T matches f then FINISH, otherwise release the lock and roll back the tuple handler	←
	Provide t 's the last fi eld f and an ack go

Table 6.3: The Protocol to Encode Polyadic Communications

texts in the context closure). Like for the π -calculus,¹ a fully abstract encoding seems very hard to achieve. The problem is that putting two encoded terms in a generic context (i.e., a context not necessarily corresponding to the encoding of any term) can break the equivalence. In our setting, consider the polyadic net $N \triangleq l :: \mathbf{in}(!x)@l.\mathbf{out}(x)@l$; in Chapter 5 (Section 5) we proved that $N \cong_{\mu K} l :: \mathbf{nil}$. However, $\langle N \rangle \not\equiv_{cK} \langle l :: \mathbf{nil} \rangle$ because of, e.g., context $[-] \parallel l :: \langle l \rangle \langle 2 \rangle$ that provides a link to an ‘unfair’ tuple handler (actually, it provides a non-restricted locality and the handler only provides the length of a tuple but not its fields). Indeed, the protocol of Table 6.3 cannot succeed because N gets blocked in Q^1 since no go will be ever produced at l .

We believe that, by relying on sophisticated typing theories (like, e.g., in [152]) to consider in the context closure only those contexts that do not violate the exchange protocol implemented by the encoding, a (restricted) fully abstraction result does hold. However, as we have already said in Chapter 3, it seems us unreasonable for a tuple-based language to assume that the repository of a node contains only data (i.e. tuples) of the same kind (i.e. with the same shape). So, even if theoretically possible, fully abstraction (w.r.t. an equivalence that is a congruence) would be in contrast with the principles underlying the tuple-space paradigm.

We now give the theoretical results. They rely on some preliminary steps,

¹[152] shows that Milner’s encoding (sketched before) is *not* fully abstract w.r.t. bisimulation.

describing the operational correspondence between polyadic nets and their encoded monadic nets.

Proposition 6.3.1 *The following facts hold.*

1. If $N \xrightarrow{\text{nil} @ l} N'$ then $\langle N \rangle \xrightarrow{\text{nil} @ l} \langle N' \rangle$; viceversa, if $\langle N \rangle \xrightarrow{\text{nil} @ l} M$ then $N \xrightarrow{\text{nil} @ l} N'$ and $M \triangleq \langle N' \rangle$.
2. If $N \xrightarrow{(\bar{\nu}l) \langle t \rangle @ l} N'$ then $\langle N \rangle \xrightarrow{(\nu l') \langle l' \rangle @ l} (\bar{\nu}l)(\langle N' \rangle \parallel l :: \mathbf{nil} \parallel l' :: R_P(t))$; viceversa, if $\langle N \rangle \xrightarrow{(\nu l') \langle l' \rangle @ l} M$ then $N \xrightarrow{(\bar{\nu}l) \langle t \rangle @ l} N'$ and $M \equiv (\bar{\nu}l)(\langle N' \rangle \parallel l :: \mathbf{nil} \parallel l' :: R_P(t))$.
3. If $N \xrightarrow{\triangleright l} N'$ then $\langle N \rangle \xrightarrow{\triangleright l} \langle N' \rangle$. Viceversa, if $\langle N \rangle \xrightarrow{\triangleright l} M$, then $N \xrightarrow{\triangleright l} N'$ and $M \succeq_{cK} \langle N' \rangle$.
4. If $N \xrightarrow{t \triangleleft l} N'$ then $\langle N \rangle \xrightarrow{l' \triangleleft l} \langle C[l'' :: Q_{l,l',X}^0(T;P)[\mathbf{in}(!x)@l.Q_{l,x,X}^0(T;P)/X]] \rangle$, where $N \equiv C[l'' :: \mathbf{in}(T)@l.P]$ for some $C[\cdot]$, l'' , T and P such that $\text{fn}(l, t) \cap \text{bn}(C[\cdot]) = \emptyset$ and $\text{match}(T, t) = \sigma$.
Viceversa, if $\langle N \rangle \xrightarrow{l' \triangleleft l} N_1$ then $N \equiv C[l'' :: \mathbf{in}(T)@l.P]$ for some $C[\cdot]$, l'' , T , and P such that $l \notin \text{bn}(C[\cdot])$. Moreover, for every t s.t. $\text{fn}(t) \cap \text{bn}(C[\cdot]) = \emptyset$ and $\text{match}(T, t) = \sigma$, it holds that $N \xrightarrow{t \triangleleft l} C[l'' :: P\sigma]$.

Proof: All the statements can be proved by induction on the inference length. The proof is long and standard, thus we omit it. ■

Lemma 6.3.2 (Preservation of Execution Steps) *If N is a polyadic net and $N \xrightarrow{\tau} N'$ then $\langle N \rangle \Rightarrow \succeq_{cK} \langle N' \rangle$.*

Proof: The proof is by induction on the length of the inference for $\xrightarrow{\tau}$. There are three base cases: when using (LTS-NEW), (LTS-SEND) and (LTS-COMM). The first one is straightforward; we now inspect the other cases.

(LTS-SEND). We have that $N \triangleq N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N'$ because $N_1 \xrightarrow{\triangleright l} N'_1$ and $N_2 \xrightarrow{\text{nil} @ l} N'_2$. In this case, we use Proposition 6.3.1.1 and .3 to conclude that $\langle N \rangle \Rightarrow \langle N'_1 \rangle \parallel \langle N'_2 \rangle \parallel l :: \langle P \rangle \equiv \langle N' \rangle$.

(LTS-COMM). We have that $N \triangleq N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N'$ because $N_1 \xrightarrow{\langle t \rangle \triangleleft l} N'_1$ and $N_2 \xrightarrow{\langle t \rangle @ l} N'_2$. Then, by using Proposition 6.3.1.2 and .4 and Proposition 6.1.6.5, we can say that $\langle N \rangle \Rightarrow (\nu l')(\langle N' \rangle \parallel l' :: \mathbf{nil}) \succeq_{cK} \langle N' \rangle$ because $l' \notin n(N)$ and, thus, $l' \notin n(\langle N' \rangle)$.

For the inductive case, we analyse the last rule used, namely (LTS-PAR), (LTS-RES) and (LTS-STRUCT). All the cases are easy. ■

Let us now consider the converse; to this aim, we need a slightly more involved. We start with a definition needed to consider the intermediate states in the execution of a communication. Recall that $l \in fl(N)$ if and only if $N \equiv N' \parallel l :: \mathbf{nil}$.

Definition 6.3.3

1. A cKLAIM net M is a partial reduct of a μ KLAIM net N whenever $N \equiv l_1 :: \mathbf{in}(T)@l_2.P \parallel l_2 :: \langle t \rangle$ and $\langle N \rangle \xrightarrow{\tau} M \xrightarrow{\tau} \succeq_{cK} \langle N \rangle$.
2. A cKLAIM net M is a partial state of a μ KLAIM net N whenever $N \equiv (\nu \bar{l})(N_1 \parallel \dots \parallel N_n \parallel \bar{N})$, $M \equiv (\nu \bar{l})(M_1 \parallel \dots \parallel M_n \parallel \langle \bar{N} \rangle)$ and for all i it holds that $fl(N_i) \subseteq fl(\bar{N})$ and that M_i is a partial reduct of N_i .

A pleasant property of partial reducts (that turns out to be crucial for the proof of Theorem 6.3.7) now follows.

Lemma 6.3.4 *If M is a partial reduct of $N \equiv l_1 :: \mathbf{in}(T)@l_2.P \parallel l_2 :: \langle t \rangle$ and $M \xrightarrow{\tau} M'$, then either M' is a partial reduct of N , or $M' \succeq_{cK} \langle N \rangle$, or $M' \succeq_{cK} l_1 :: \langle P\sigma \rangle \parallel l_2 :: \mathbf{nil}$, where $\sigma = match(T, t)$.*

Proof: By Definition 6.3.3.1 and by inspection on the possible reductions. ■

Lemma 6.3.5 (Reflection of Execution Steps) *If N is a polyadic net and $\langle N \rangle \xrightarrow{\tau} M$ then either $N \xrightarrow{\tau} N'$ and $M \succeq_{cK} \langle N' \rangle$, or M is a partial state of N .*

Proof: The proof is by induction on the length of the inference for $\langle N \rangle \xrightarrow{\tau} M$ having τ as label. There are three base cases:

(LTS-SEND). In this case it holds that $\langle N \rangle \triangleq \langle N_1 \rangle \parallel \langle N_2 \rangle \xrightarrow{\tau} M$. Then, by definition, $\langle N_1 \rangle \xrightarrow{p \cdot l} M_1$, $\langle N_2 \rangle \xrightarrow{\mathbf{nil} @ l} M_2$ and $M \triangleq M_1 \parallel M_2$. By Proposition 6.3.1.3, we know that $N_1 \xrightarrow{p \cdot l} N'_1$ and $M_1 \succeq_{cK} \langle N'_1 \rangle$. Thus, $N \xrightarrow{\tau} N'_1 \parallel N_2 \triangleq N'$ and $M \succeq_{cK} \langle N' \rangle$.

(LTS-COMM). In this case it holds that $\langle N \rangle \triangleq \langle N_1 \rangle \parallel \langle N_2 \rangle \xrightarrow{\tau} M_1 \parallel M_2 \triangleq M$ because $\langle N_1 \rangle \xrightarrow{l' \cdot l} M_1$ and $\langle N_2 \rangle \xrightarrow{\langle l' \rangle @ l} M_2$. This case is *not* possible, since no encoding of a net can directly offer a non-restricted datum.

(LTS-NEW). This case trivially falls in the first possibility of this Lemma.

For the inductive case, we reason on the last rule used in the inference.

(LTS-PAR). In this case it holds that $\langle N \rangle \triangleq \langle N_1 \rangle \parallel \langle N_2 \rangle \xrightarrow{\tau} M' \parallel \langle N_2 \rangle \triangleq M$ because $\langle N_1 \rangle \xrightarrow{\tau} M'$. By induction, either $N_1 \xrightarrow{\tau} N'_1$ and $M' \succeq_{cK} \langle N'_1 \rangle$, or M' is a partial state of N_1 . In the first case, we have that $N \xrightarrow{\tau} N'_1 \parallel N_2 \triangleq N'$ and $M \succeq_{cK} \langle N' \rangle$. In the second case, M is a partial state of N , by definition.

(LTS-RES). We now identify two possible sub-cases:

- $\langle N \rangle \triangleq \langle (\nu l)N_1 \rangle \triangleq (\nu l)\langle N_1 \rangle \xrightarrow{\tau} (\nu l)M_1 \triangleq M$ because $\langle N_1 \rangle \xrightarrow{\tau} M_1$. This case easily follows by induction.
- $\langle N \rangle \triangleq (\nu l)M_1 \xrightarrow{\tau} (\nu l)M_2 \triangleq M$ because $M_1 \xrightarrow{\tau} M_2$ but M_1 is not the encoding of any polyadic net. In this case, locality l is fresh for N and has been introduced by the encoding. It is then easy to see that l is the reference for a datum located in a node of N , i.e. $M_1 \triangleq l_1 :: \langle l \rangle \parallel l :: R_l(t)$, and $N \triangleq l_1 :: \langle t \rangle$. But then no τ -step can be performed by M_1

(LTS-STRUCT). In this case we have that $\langle N \rangle \equiv M_1 \xrightarrow{\tau} M_2 \equiv M$. Let $A_N \triangleq \text{bn}(\langle N \rangle) - n(N)$ be the (restricted) names introduced by the encoding; we then proceed by induction on k , the number of names in A_N touched by rules (STR-RCOM) and (STR-EXT) in deriving $\langle N \rangle \equiv M_1$. We shall refer to this latter induction as the *internal* induction, while the induction on the number of rules used to infer $\xrightarrow{\tau}$ will be called the *external* one.

Base. If $k = 0$, then we can claim that $M_1 \triangleq \langle N'' \rangle$ for some $N'' \equiv N$ (this can be proved by an easy induction on the length of $\langle N \rangle \equiv M_1$). The thesis holds by using this fact and a straightforward external induction.

Induction. Let $l \in A_N$. Then $\langle N \rangle \triangleq C[(\nu l)(l' :: \langle l \rangle \parallel l :: R_l(t))]$ for some t and l' . By using Lemma 5.2.6 and a simple analysis over the definition of the involved processes, it can only be one of the following cases:

- $C[\mathbf{0}] \xrightarrow{\tau} C'[\mathbf{0}]$ and $M \equiv C'[(\nu l)(l' :: \langle l \rangle \parallel l :: R_l(t))]$. But then $\langle N \rangle \triangleq C[(\nu l)(l' :: \langle l \rangle \parallel l :: R_l(t))] \xrightarrow{\tau} M$ can be inferred without touching l with rules (STR-RCOM) and (STR-EXT) Hence, by internal induction, we can conclude.
- $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ where $H \xrightarrow{l'} H'$, $l' \notin \text{bn}(C_2[\cdot])$ and $M \equiv C_1[(\nu l)(C_2[l' :: \langle l \rangle \parallel l :: R_l(t)] \parallel H')]$. Like in the previous case, $\langle N \rangle \triangleq C[(\nu l)(l' :: \langle l \rangle \parallel l :: R_l(t))] \xrightarrow{\tau} M$ can be inferred without touching l with rules (STR-RCOM) and (STR-EXT) Hence, by internal induction, we can conclude.
- $C[\cdot] \triangleq C_1[C_2[\cdot] \parallel H]$ where $H \xrightarrow{l'} H'$ and $M \equiv C_1[C_2[(\nu l)(l' :: \mathbf{nil} \parallel l :: R_l(t)) \parallel l' :: P' \parallel l' :: \mathbf{nil} \parallel \mathcal{E}[l' :: \mathbf{nil}]]]$. By Proposition 6.3.1.4, $H \triangleq \mathcal{E}[l' :: \mathbf{rec} X.\mathbf{in}(!x)@l'.Q_{l',x,X}^0(T;P)]$, for some $\mathcal{E}[\cdot]$, l' , T and P , where context $\mathcal{E}[\cdot]$ does not bind l and l' ; moreover, $H' \equiv \mathcal{E}[l' :: P']$, where $P' \triangleq Q_{l',l,X}^0(T;P)[\mathbf{rec} X.\mathbf{in}(!x)@l'.Q_{l',x,X}^0(T;P)/X]$. Thus,

$$M \equiv C_1[C_2[(\nu l)(l' :: \mathbf{nil} \parallel l :: R_l(t)) \parallel l' :: P' \parallel l' :: \mathbf{nil} \parallel \mathcal{E}[l' :: \mathbf{nil}]]]$$

Now, we have that

$$N \equiv \mathcal{D}[l' :: \langle t \rangle \parallel l' :: \mathbf{in}(T)@l'.P] \parallel l' :: \mathbf{nil} \parallel \mathcal{F}[l' :: \mathbf{nil}]$$

for $\mathcal{E}[\cdot] \triangleq \langle \mathcal{F}[\cdot] \rangle$ and $C_1[C_2[\cdot]] \triangleq \langle \mathcal{D} \rangle[\cdot]$. By definition, we have that M is a partial state of N . ■

We then need a Lemma that relates the behaviour (both the barbs and the reductions) of a partial state M of N to the behaviour of the encoding of N .

Lemma 6.3.6 *Let M be a partial state of N .*

1. *If $N \downarrow l$ then $M \downarrow l$; moreover, if $N \xrightarrow{\tau} N'$ then $M \Rightarrow \langle N' \rangle$.*
2. *If $M \downarrow l$ then $N \downarrow l$.*
3. *If $M \xrightarrow{\tau} M'$, then $N \xrightarrow{\hat{\tau}} N'$ for some N' such that $M' \succeq_{cK}^{\text{tr}} M''$, where M'' is a partial state of N' .*

Proof:

1. By exploiting Proposition 6.3.1.3 and Lemma 6.3.2 respectively, this case is simple, once noticed that $M \Rightarrow \succeq_{cK} \langle N \rangle$ (by definition of partial states).
2. By definition, $N \equiv (\nu \bar{l})(N_1 \parallel \dots \parallel N_n \parallel \bar{N})$ and $M \equiv (\nu \bar{l})(M_1 \parallel \dots \parallel M_n \parallel \langle \bar{N} \rangle)$, where M_i is a partial reduct of N_i . By construction, we know that M_i can only host data on restricted locations; thus, $M_i \not\downarrow l$. This implies that $\langle \bar{N} \rangle \downarrow l$ and $l \notin \bar{l}$; because of Proposition 6.3.1.2, $\bar{N} \downarrow l$ and hence $N \downarrow l$.
3. We know that $N \equiv (\nu \bar{l})(N_1 \parallel \dots \parallel N_n \parallel \bar{N})$, $M \equiv (\nu \bar{l})(M_1 \parallel \dots \parallel M_n \parallel \langle \bar{N} \rangle)$, and for all $i = 1, \dots, n$ it holds that $fl(N_i) \subseteq fl(\bar{N})$ and that M_i is a partial reduct of N_i . The crucial observation is that there are only two possible cases:

$$M_i \xrightarrow{\tau} M'_i \text{ and } M' \equiv (\nu \bar{l})(M_1 \parallel \dots \parallel M_{i-1} \parallel M'_i \parallel M_{i+1} \parallel \dots \parallel M_n \parallel \langle \bar{N} \rangle).$$

By Lemma 6.3.4 we have three possible sub-cases:

- (a) M'_i is a partial reduct of N_i : in this case, M' is still a partial state of N . By construction, $(N, M') \in \mathfrak{R}$.
- (b) $M'_i \succeq_{cK} \langle N_i \rangle$: by contextuality of \lesssim_{cK} , it holds that $M' \succeq_{cK} (\nu \bar{l})(M_1 \parallel \dots \parallel M_{i-1} \parallel M_{i+1} \parallel \dots \parallel M_n \parallel \langle N_i \parallel \bar{N} \rangle) \triangleq M''$. Now, M'' is a partial state of N and, hence, $(N, M') \in \mathfrak{R}$ up-to $\lesssim_{\mu K}$.
- (c) $N_i \equiv l_1 :: \mathbf{in}(T)@l_2.P \parallel l_2 :: \langle t \rangle$ and $M'_i \succeq_{cK} l_1 :: \langle P\sigma \rangle \parallel l_2 :: \mathbf{nil} \triangleq N'$, where $\sigma = \text{match}(T, t)$: in this case, we can consider $N_i \xrightarrow{\tau} l_1 :: P\sigma \parallel l_2 :: \mathbf{nil} \triangleq N'_i$ and have that $M'_i \succeq_{cK} \langle N'_i \rangle$. Thus, $N \xrightarrow{\tau} (\nu \bar{l})(N_1 \parallel \dots \parallel N_{i-1} \parallel N_{i+1} \parallel \dots \parallel N_n \parallel N'_i \parallel \bar{N})$ and $M' \succeq_{cK} (\nu \bar{l})(M_1 \parallel \dots \parallel M_{i-1} \parallel M_{i+1} \parallel \dots \parallel M_n \parallel \langle N'_i \parallel \bar{N} \rangle) \triangleq M''$. Since M'' is a partial state for N , we have that $(N', M') \in \mathfrak{R}$ up-to $\lesssim_{\mu K}$.

$$\langle \bar{N} \rangle \xrightarrow{\tau} \bar{M} \text{ and } M' \equiv (\nu \bar{l})(M_1 \parallel \dots \parallel M_n \parallel \bar{M}).$$

By Lemma 6.3.5, we have two possible sub-cases:

- (a) $\bar{N} \xrightarrow{\tau} \bar{N}'$ and $\bar{M} \succeq_{cK} \bar{N}'$: in this case, $N \xrightarrow{\tau} (\nu\bar{l})(N_1 \parallel \dots \parallel N_n \parallel \bar{N}')$ $\triangleq N'$ and $M' \succeq_{cK} (\nu\bar{l})(M_1 \parallel \dots \parallel M_n \parallel \langle \bar{N}' \rangle)$; thus, $(N', M') \in \mathfrak{R}$ up-to $\lesssim_{\mu K}$.
- (b) \bar{M} is a partial state of \bar{N} : by definition, we have that $\bar{N} \equiv (\nu\bar{l}')(H_1 \parallel \dots \parallel H_h \parallel \bar{H})$, $\bar{M} \equiv (\nu\bar{l}')(K_1 \parallel \dots \parallel K_h \parallel \langle \bar{H} \rangle)$, and for all $j = 1, \dots, h$ it holds that $f_l(H_j) \subseteq f_l(\bar{H})$ and that K_j is a partial reduct of H_j . Thus, $N \equiv (\nu\bar{l}, l')(N_1 \parallel \dots \parallel N_n \parallel H_1 \parallel \dots \parallel H_h \parallel \bar{H})$ and $M' \equiv (\nu\bar{l}, l')(M_1 \parallel \dots \parallel M_n \parallel K_1 \parallel \dots \parallel K_h \parallel \langle \bar{H} \rangle)$, where M_i is a partial reduct of N_i and K_j is a partial reduct of H_j . Moreover, we also have that $f_l(H_j) \subseteq f_l(\bar{H})$ (by definition) and that $f_l(N_i) \subseteq f_l(\bar{H})$ (this easily follows from $f_l(N_i) \subseteq f_l(\bar{N})$ and by definition of \bar{N}). Thus, M' is a partial state of N ; this suffices to conclude $(N, M') \in \mathfrak{R}$. ■

To conclude this section, we can formulate a restricted full abstraction result, by following [13]. In particular, we shall consider for full abstraction the translated barbed congruence.

Theorem 6.3.7 (Full Abstraction w.r.t. Translated Barbed Congruence)

$N \cong_{\mu K} M$ if and only if $\langle N \rangle \cong_{cK}^{\text{tr}} \langle M \rangle$.

Proof: For the ‘if’ direction, it suffices to prove that relation \mathfrak{R} defined as follows

$$\begin{aligned} \mathfrak{R} &\triangleq \mathfrak{R}_1 \cup \mathfrak{R}_2 \cup \mathfrak{R}_3 \\ \mathfrak{R}_1 &\triangleq \{(N, M) : \langle N \rangle \cong_{cK} \langle M \rangle\} \\ \mathfrak{R}_2 &\triangleq \{(N, M) : \exists \bar{M}. \langle N \rangle \cong_{cK} \bar{M} \wedge \bar{M} \text{ partial state of } M\} \\ \mathfrak{R}_3 &\triangleq \{(N, M) : \exists \bar{N}. \bar{N} \cong_{cK} \langle M \rangle \wedge \bar{N} \text{ partial state of } N\} \end{aligned}$$

is barb preserving, reduction closed (up-to $\lesssim_{cK}^{\text{tr}}$) and closed under translated contexts (again, up-to $\lesssim_{cK}^{\text{tr}}$). Notice that \mathfrak{R}_1 is symmetric, while \mathfrak{R}_2 and \mathfrak{R}_3 are mutually symmetric; thus, \mathfrak{R} is symmetric. We pick up $(N, M) \in \mathfrak{R}$ and reason by case analysis on whether $(N, M) \in \mathfrak{R}_1$, $(N, M) \in \mathfrak{R}_2$ or $(N, M) \in \mathfrak{R}_3$.

1. Let $N \downarrow l$ (the case for $M \downarrow l$ is similar). By definition and Proposition 6.3.1.2, we have that $\langle N \rangle \downarrow l$ that implies $\langle M \rangle \downarrow l$, i.e. $\langle M \rangle \mapsto^* M' \downarrow l$. According to Lemma 6.3.5, we have two possibilities:

- (a) $M \mapsto^* M''$ and $M' \succeq_{cK}^{\text{tr}} \langle M'' \rangle$. In this case, by definition of \succeq_{cK}^{tr} , we have that $\langle M'' \rangle \downarrow l$ and hence $M \downarrow l$.
- (b) M' is a partial reduct of M . By Lemma 6.3.6.2, $M \downarrow l$.

Now, let $N \mapsto N'$; then, $\langle N \rangle \mapsto^* \bar{N} \succeq_{cK}^{\text{tr}} \langle N' \rangle$. By definition of reduction closure, $\langle M \rangle \mapsto^* \bar{M}$ and $\bar{N} \cong_{cK}^{\text{tr}} \bar{M}$. According to Lemma 6.3.5, we have two possibilities:

- (a) $M \mapsto^* M'$ and $\bar{M} \succeq_{cK}^{\text{tr}} \langle M' \rangle$. In this case is simple: because of Proposition 6.1.2 and by transitivity, we can obtain $\langle N' \rangle \cong_{cK}^{\text{tr}} \langle M' \rangle$ and, hence, $(N', M') \in \mathfrak{R}_1$.
- (b) M' is a partial reduct of M . By construction, $(N', M) \in \mathfrak{R}_2$ (notice that, if the starting move was from M instead of being from N , the inclusion would have been in \mathfrak{R}_3).

We are left with context closure; this case is simple because, if we take any μKLAIM context $C[\cdot]$, by definition of \mathfrak{R}_1 and because $\langle C[\cdot] \rangle = \langle C[\cdot] \rangle$, we have that $(C[N], C[M]) \in \mathfrak{R}_1$.

2. Let $(N, M) \in \mathfrak{R}_2$; by definition, there exists a partial reduct of M , \bar{M} , such that $\langle N \rangle \cong_{cK} \bar{M}$. Let us start with $N \downarrow l$; hence, $\bar{M} \Downarrow l$, i.e. $\bar{M} \mapsto^* \bar{M}' \downarrow l$. Now, by using Lemma 6.3.6.3, we have that $M \mapsto^* M'$ for some M' such that $\bar{M}' \succeq_{cK}^{\text{tr}} \bar{M}''$, where \bar{M}'' is a partial state of M' . By definition of $\lesssim_{cK}^{\text{tr}}$, we have $\bar{M}'' \downarrow l$ and, by Lemma 6.3.6.2, $M' \downarrow l$; this suffices to conclude $M \Downarrow l$.

Now, let $N \mapsto N'$; then, by Lemma 6.3.2, $\langle N \rangle \mapsto^* \bar{N} \succeq_{cK}^{\text{tr}} \langle N' \rangle$. By reduction closure, $\bar{M} \mapsto^* \bar{M}'$ and $\bar{N} \cong_{cK}^{\text{tr}} \bar{M}'$, that implies $\langle N' \rangle \cong_{cK}^{\text{tr}} \bar{M}'$. By Lemma 6.3.6.3, $M \mapsto^* M'$ and \bar{M}' is an expansion of a partial state of M' , say \bar{M}'' . By Proposition 6.1.2 and transitivity, $\langle N' \rangle \cong_{cK}^{\text{tr}} \bar{M}''$; this suffices to conclude that $(N', M') \in \mathfrak{R}_2$.

We are left with dealing with context closure; by definition, we have that $\langle C[\bar{M}] \rangle \cong_{cK}^{\text{tr}} \langle C[N] \rangle$. If we prove that $\langle C[\bar{M}] \rangle$ is a partial state of $C[M]$, we can conclude the desired $(C[N], C[M]) \in \mathfrak{R}_2$. Since \bar{M} is a partial state of M , we have that $M \equiv (\nu l)(M_1 \parallel \dots \parallel M_n \parallel \hat{M})$, $\bar{M} \equiv (\nu l)(\bar{M}_1 \parallel \dots \parallel \bar{M}_n \parallel \langle \hat{M} \rangle)$, and for all $i = 1, \dots, n$ it holds that $fl(M_i) \subseteq fl(\hat{M})$ and that \bar{M}_i is a partial reduct of M_i . Let $l' = bn(\langle C[\cdot] \rangle) \cap fn(\bar{M}_1, \dots, \bar{M}_n) = bn(C[\cdot]) \cap fn(M_1, \dots, M_n)$; then, $C[\cdot] \equiv (\nu l')\mathcal{D}[\cdot]$ and $\langle C[\cdot] \rangle \equiv (\nu l')\langle \mathcal{D}[\cdot] \rangle$. Thus, $C[M] \equiv (\nu l')(M_1 \parallel \dots \parallel M_n \parallel \mathcal{D}[\hat{M}])$ and $\langle C[\bar{M}] \rangle \equiv (\nu l')(\bar{M}_1 \parallel \dots \parallel \bar{M}_n \parallel \langle \mathcal{D}[\bar{M}] \rangle)$; clearly, $\langle C[\bar{M}] \rangle$ is a partial state of $C[M]$.

3. Finally, let $(N, M) \in \mathfrak{R}_3$; by definition, there exists a partial reduct of N , \bar{N} , such that $\langle M \rangle \cong_{cK} \bar{N}$. Let us start with barb preservation and let $N \downarrow l$; by Lemma 6.3.6.1, $\bar{N} \Downarrow l$, i.e. $\bar{N} \mapsto^* \bar{N}' \downarrow l$. Now, $\langle M \rangle \Downarrow l$ that, like in case 1. above, implies $M \Downarrow l$, as required.

Now, let $N \mapsto N'$; then, Lemma 6.3.6.1, $\bar{N} \mapsto^* \bar{N}' \succeq_{cK}^{\text{tr}} \langle N' \rangle$. By reduction closure, $\langle M \rangle \mapsto^* \bar{M}$ and $\bar{N}' \cong_{cK}^{\text{tr}} \bar{M}$, that implies $\langle N' \rangle \cong_{cK}^{\text{tr}} \bar{M}$. By Lemma 6.3.5, we have two possibilities:

- (a) $M \mapsto^* M'$ and $\bar{M} \succeq_{cK}^{\text{tr}} \langle M' \rangle$. By Proposition 6.1.2 and transitivity, we can conclude that $(N', M') \in \mathfrak{R}_1$.
- (b) $M \mapsto^* M'$ and \bar{M} is a partial state of M' . By construction, $(N', M') \in \mathfrak{R}_2$.

Context closure is proved like in case 2. above.

Encoding Nets:	
$\llbracket \mathbf{0} \rrbracket \triangleq \mathbf{0}$	$\llbracket (\nu l)N \rrbracket \triangleq (\nu l)\llbracket N \rrbracket$
$\llbracket N_1 \parallel N_2 \rrbracket \triangleq \llbracket N_1 \rrbracket \parallel \llbracket N_2 \rrbracket$	$\llbracket l :: C \rrbracket \triangleq l :: \llbracket C \rrbracket_l$
Encoding Components:	
$\llbracket \langle l' \rangle \rrbracket_u \triangleq \langle l' \rangle$	$\llbracket C_1 \mid C_2 \rrbracket_u \triangleq \llbracket C_1 \rrbracket_u \mid \llbracket C_2 \rrbracket_u$
$\llbracket \mathbf{nil} \rrbracket_u \triangleq \mathbf{nil}$	$\llbracket X \rrbracket_u \triangleq X$
$\llbracket \mathbf{eval}(Q)@u'.P \rrbracket_u \triangleq \mathbf{eval}(\llbracket Q \rrbracket_{u'})@u'.\llbracket P \rrbracket_u$	$\llbracket \mathbf{rec} X.P \rrbracket_u \triangleq \mathbf{rec} X.\llbracket P \rrbracket_u$
$\llbracket \mathbf{out}(u_2)@u_1.P \rrbracket_u \triangleq \mathbf{eval}(\mathbf{out}(u_2))@u_1.\llbracket P \rrbracket_u$	$\llbracket \mathbf{new}(l).P \rrbracket_u \triangleq \mathbf{new}(l).\llbracket P \rrbracket_u$
$\llbracket \mathbf{in}(T)@u'.P \rrbracket_u \triangleq \mathbf{eval}(\mathbf{in}(T).\mathbf{eval}(\llbracket P \rrbracket_u)@u)@u'$	

Table 6.4: Encoding cKLAIM in LCKLAIM

We are left with the ‘only if’ direction; this can be done similarly to the ‘if’ direction. We leave the details to the reader. ■

6.4 cKLAIM VS LCKLAIM

The encoding of cKLAIM into LCKLAIM is given in Table 6.4. The only relevant cases are those for the translation of actions **in** and **out** of cKLAIM. In the first case, a remote action **out** is replaced with a migration to the target locality and a local action. In the second case, a remote action **in** is replaced with a migration to the target locality, a local **in** and a migration back to the original node. The subscript u in $\llbracket \cdot \rrbracket_u$ is needed to keep track of the original node where the result of the (remote) action will be sent back.

To the best of our knowledge, this is the first result clearly showing that remote (input and output) operations do not add expressiveness to a distributed language with code mobility. Having remote operations simplifies programming. Having only migrations allows for finer dynamic checks against incoming agents (see, e.g., [129] or the theory presented in Chapters 3 and 4).

In order to carry on the proofs, we introduce an auxiliary notion. We define a function between LCKLAIM nets, $nrm_L(\cdot)$, called the *normalisation w.r.t. a set of localities* L , as follows

$$\begin{aligned}
nrm_L(N_1 \parallel N_2) &\triangleq nrm_L(N_1) \parallel nrm_L(N_2) \\
nrm_L((\nu l)N) &\triangleq nrm_{L \cup \{l\}}(N) \\
nrm_L(l :: C_1 \mid C_2) &\triangleq nrm_L(l :: C_1) \parallel nrm_L(l :: C_2) \\
nrm_L(l :: \langle \cdot \rangle) &\triangleq l :: \langle \cdot \rangle
\end{aligned}$$

$$nrm_L(l :: P) \triangleq \begin{cases} l :: P' \parallel l' :: Q & \text{if } P = a.P' \text{ and } a = \mathbf{eval}(Q)@l' \text{ and } l' \in L \\ & \text{and } Q = \mathbf{in}(T).\mathbf{eval}(\llbracket P \rrbracket_l)@l \\ l :: P & \text{if } P \neq _|_ \end{cases}$$

Essentially, the normalisation of an encoding replaces all the encodings of actions **in** occurring at top level (i.e., as the first action of a process) with the net resulting from the execution of their first actions (i.e., the migration over the locality target of the **in**), provided that this execution is possible (i.e., the target locality of the input exists in the net).

Now, it is easy to prove the following Proposition. For the sake of readability, we write $nrm_L(\llbracket N \rrbracket)$ as $\llbracket N \rrbracket_L$ and $\llbracket N \rrbracket_{\mathcal{R}(N)}$ as $\{\!\{ N \}\!\}$.

Proposition 6.4.1 *Let N be a cKLAIM net and M be a lCKLAIM net. Then*

1. $fl(M) = fl(\llbracket M \rrbracket_L)$, whenever $L \subseteq fl(M)$
2. $M \succeq_{lcK} nrm_L(M)$, whenever $L \subseteq fl(M)$
3. $\llbracket N \rrbracket_L \parallel l :: \mathbf{nil} \succeq_{lcK} \llbracket N \rrbracket_{L \cup \{l\}} \parallel l :: \mathbf{nil}$
4. $\llbracket N \rrbracket \succeq_{lcK} \{\!\{ N \}\!\}$

Lemma 6.4.2 *Let N be a cKLAIM net and $fl(N) \subseteq L$. Then*

1. if $N \xrightarrow{(\bar{v}l)I@l} N'$ then $\llbracket N \rrbracket_L \xrightarrow{(\bar{v}l)I@l} \llbracket N' \rrbracket_{L \cup \{\bar{l}\}}$
2. if $N \xrightarrow{\triangleright l} N'$ then $\llbracket N \rrbracket_L \xrightarrow{\triangleright l} \llbracket N' \rrbracket_L$
3. if $N \xrightarrow{l_2 \triangleleft l_1} N'$ then either $\llbracket N \rrbracket_L \xrightarrow{l_2 \triangleleft l_1} \llbracket N' \rrbracket_L$, or $\llbracket N \rrbracket_L \equiv C[l :: \mathbf{eval}(\mathbf{in}(T).\mathbf{eval}(\llbracket P \rrbracket_l)@l)@l_1]$ where $match(T, l_2) = \sigma$, $l_1 \notin L$, $\{l_1, l_2\} \cap bn(C[\cdot]) = \emptyset$ and $\llbracket N' \rrbracket_L \equiv nrm_L(C[l :: \llbracket P\sigma \rrbracket_l])$
4. if $\llbracket N \rrbracket_L \xrightarrow{(\bar{v}l)I@l} N'$, then $N \xrightarrow{(\bar{v}l)I@l} N''$ and $N' \succeq_{lcK} \llbracket N'' \rrbracket_{L \cup \{\bar{l}\}}$
5. if $\llbracket N \rrbracket_L \xrightarrow{l_2 \triangleleft l_1} N'$, then $N \xrightarrow{l_2 \triangleleft l_1} N''$ and $N' \succeq_{lcK} \llbracket N'' \rrbracket_L$
6. if $\llbracket N \rrbracket_L \xrightarrow{\triangleright l} N'$ then
 - (a) either $N \xrightarrow{\triangleright l} N''$ and $N' \succeq_{lcK} \llbracket N'' \rrbracket_L$
 - (b) or $N \equiv C[l' :: \mathbf{in}(T)@l.Q]$ for $l \notin bn(C[\cdot]) \cup L$ and $N' \equiv \llbracket C[l' :: \mathbf{nil} \parallel l :: \mathbf{in}(T).\mathbf{eval}(\llbracket P \rrbracket_l)@l] \rrbracket_L$

Proof: All the statements are proved by induction over the length of the inference used to derive the transition; the proof is standard. \blacksquare

Lemma 6.4.3 (Operational Correspondence) *Let N be a cKLAIM net. Then*

1. if $N \xrightarrow{\tau} N'$, then $\llbracket N \rrbracket \Rightarrow \succeq_{lcK} \llbracket N' \rrbracket$
2. if $\llbracket N \rrbracket \xrightarrow{\tau} N'$, then $N \xrightarrow{\tau} N''$ and $N' \succeq_{lcK} \llbracket N'' \rrbracket$

Proof: Both the claims are proved by induction on the inference length. The inductive steps are easy: they rely on the fact that \preceq_{lcK} is a pre-congruence and on the observation that $N \equiv M$ implies $\llbracket N \rrbracket \equiv \llbracket M \rrbracket$. Thus, we only present the base cases for both the claims.

In the first claim, the τ -step can be inferred by using rules (LTS-NEW), (LTS-SEND) or (LTS-COMM). The first case is simple; hence, let us consider the other two.

(LTS-SEND): in this case, $N \triangleq N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N'$, where $N_1 \xrightarrow{\triangleright l} N'_1$ and $N_2 \xrightarrow{\text{nil} @ l} N'_2$. The key observation is that $f(N_i) \subseteq f(N) = f(N'_1 \parallel N'_2) = f(N')$; let us call L the set $f(N)$. By Lemma 6.4.2.1 and .2, we have that $\llbracket N_2 \rrbracket_L \xrightarrow{\text{nil} @ l} \llbracket N'_2 \rrbracket_L$ and $\llbracket N_1 \rrbracket_L \xrightarrow{\triangleright l} \llbracket N'_1 \rrbracket_L$. Thus, $\llbracket N \rrbracket \Rightarrow \llbracket N'_1 \parallel N'_2 \rrbracket \triangleq \llbracket N' \rrbracket$.

(LTS-COMM): in this case, $N \triangleq N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N'$, where $N_1 \xrightarrow{l_2 \triangleleft l_1} N'_1$ and $N_2 \xrightarrow{\langle l_2 \rangle @ l_1} N'_2$. Again, we have that $f(N_i) \subseteq f(N) = f(N')$; let us call L the set $f(N)$. By Lemma 6.4.2.1 we have that $\llbracket N_2 \rrbracket_L \xrightarrow{\langle l_2 \rangle @ l_1} \llbracket N'_2 \rrbracket_L$. Moreover, according to Lemma 6.4.2.3, we have two cases. The case for $\llbracket N_1 \rrbracket_L \xrightarrow{l_2 \triangleleft l_1} \llbracket N'_1 \rrbracket_L$ is simple. The case when $\llbracket N_1 \rrbracket_L \equiv C[l :: \text{eval}(\text{in}(T).\text{eval}(\llbracket P \rrbracket)_l) @ l_1]$ cannot occur. Otherwise, we would have that $l_1 \notin L$; but this cannot be the case since, by Proposition 5.2.3.2, we know that $N_2 \equiv N'_2 \parallel l_1 :: \langle l_2 \rangle$. Hence $l_1 \in f(N_2) \subseteq f(N) \triangleq L$.

The second claim is similar. We reason by case analysis on the possible base cases. The case for rule (LTS-NEW) is simple and we only inspect the other two ones. In what follows, we let L to be $f(N)$.

(LTS-SEND): in this case, $\llbracket N \rrbracket \triangleq M_1 \parallel M_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N'$, where $M_1 \triangleq \llbracket N_1 \rrbracket_L \xrightarrow{\triangleright l} N'_1$ and $M_2 \triangleq \llbracket N_2 \rrbracket_L \xrightarrow{\text{nil} @ l} N'_2$. By Lemma 6.4.2.1 we have that $N_2 \xrightarrow{\text{nil} @ l} N'_2$ and $N'_2 \succeq_{lcK} \llbracket N'_2 \rrbracket_L$. We now identify two possible sub-cases:

- (a) $C = \llbracket P \rrbracket_l$. Then, by Lemma 6.4.2.6(a) we have that $N_1 \xrightarrow{\triangleright l} N'_1$ and $N'_1 \succeq_{lcK} \llbracket N'_1 \rrbracket_L$. Thus, $N \triangleq N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N''$ and $N' \succeq_{lcK} \llbracket N'_1 \parallel N'_2 \rrbracket \triangleq \llbracket N'' \rrbracket$.
- (b) $C \triangleq \text{in}(T).\text{eval}(\llbracket Q \rrbracket_{l'}) @ l'$. By Lemma 6.4.2.6(b) we know that $N_1 \equiv C[l' :: \text{in}(T) @ l.Q]$, with $l' \notin \text{bn}(C[\cdot]) \cap L$. This case cannot occur because, by Proposition 5.2.3.1, we know that $N_2 \equiv N'_2 \parallel l :: \text{nil}$; hence, $l \in L$.

(LTS-COMM): in this case, $\{\!\!\{N\}\!\!\} \triangleq M_1 \parallel M_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N'$, where $M_1 \triangleq \{\!\!\{N_1\}\!\!\}_L \xrightarrow{l_2 \triangleleft l_1} N'_1$ and $M_2 \triangleq \{\!\!\{N_2\}\!\!\}_L \xrightarrow{\langle l_2 \rangle @ l_1} N'_2$. By Lemma 6.4.2.4 we have that $N_2 \xrightarrow{\langle l_2 \rangle @ l_1} N'_2$ and $N'_2 \succeq_{lcK} \{\!\!\{N'_2\}\!\!\}_L$; by Lemma 6.4.2.5 we have that $N_1 \xrightarrow{l_2 \triangleleft l_1} N'_1$ and $N'_1 \succeq_{lcK} \{\!\!\{N'_1\}\!\!\}_L$. Thus, $N \triangleq N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N'$ and $N' \succeq_{lcK} \{\!\!\{N'_1\}\!\!\}_L \parallel \{\!\!\{N'_2\}\!\!\}_L \triangleq \{\!\!\{N''\}\!\!\}$. ■

Theorem 6.4.4 *Let N be a cKLAIM net. Then, $N \cong_{cK} \{\!\!\{N\}\!\!\}$.*

Proof: By Lemma 6.1.5.1, it suffices to prove that

$$\mathfrak{R} \triangleq \{(C[N], C[\{\!\!\{N\}\!\!\}]) : N \text{ is a cKLAIM net and } C[\cdot] \text{ is a cKLAIM context}\}$$

is barb preserving, reduction closed (up-to \lesssim_{cK}) and context closed. Clearly, we consider here the restriction of $\cong_{\mu K}$ and $\lesssim_{\mu K}$ to cKLAIM nets; all the proofs developed for μ KLAIM can be faithfully rephrased to deal with the sub-relations containing only cKLAIM nets.

Barb preservation and context closure are simple. Let us consider $C[N] \mapsto \bar{N}$. According to Lemma 5.2.6, we have six possible sub-cases:

1. $N \mapsto N'$ and $\bar{N} \equiv C[N']$. Because of Lemma 6.4.3.1, we know that $\{\!\!\{N\}\!\!\} \Rightarrow \succeq_{cK} \{\!\!\{N'\}\!\!\}$; thus, we can conclude up-to \lesssim_{cK} .
2. $C[\cdot] \mapsto C'[\cdot]$ and $\bar{N} \equiv C'[N]$. This case is trivial.
3. $N \xrightarrow{\triangleright l} N'$, $C[\cdot] \equiv C[\cdot \parallel l :: \mathbf{nil}]$ and $\bar{N} \equiv C[N']$. Because of Lemma 6.4.2.2, we know that $\{\!\!\{N\}\!\!\} \xRightarrow{\triangleright l} \succeq_{cK} \{\!\!\{N'\}\!\!\}$ and we can easily conclude.
4. $N \xrightarrow{\mathbf{nil} @ l} N'$, $C[\cdot] \equiv C'[\cdot \parallel H]$, $H \xrightarrow{\triangleright l} H'$ and $\bar{N} \equiv C'[N' \parallel L']$. This case relies on Lemma 6.4.2.1 and is simple.
5. $N \xrightarrow{l' \triangleleft l} N'$, $C[\cdot] \equiv C'[\cdot \parallel l :: \langle l' \rangle]$ and $\bar{N} \equiv C'[N']$. The proof relies on Lemma 6.4.2.3 to show that $C[\{\!\!\{N\}\!\!\}] \Rightarrow C'[\{\!\!\{N'\}\!\!\}]$; now it is easy to conclude.
6. $N \xrightarrow{(\nu \bar{l}) \langle l' \rangle @ l} N'$, $C[\cdot] \equiv C'[\cdot \parallel H]$, $H \xrightarrow{l' \triangleleft l} H'$ and $\bar{N} \equiv C'[(\nu \bar{l})(N' \parallel H')]$. This case relies on Lemma 6.4.2.1 and is simple.

To conclude, let us consider $C[\{\!\!\{N\}\!\!\}] \mapsto \bar{N}$. According to Lemma 5.2.6, we still have six possible sub-cases:

1. $\{\!\!\{N\}\!\!\} \mapsto N'$ and $\bar{N} \equiv C[N']$. Because of Lemma 6.4.3.3, we know that $N \mapsto N''$ and $N' \succeq_{lcK} \{\!\!\{N\}\!\!\}$; this suffices to conclude up-to \lesssim_{cK} (indeed, by considering both N' and $\{\!\!\{N\}\!\!\}$ as cKLAIM nets, we have that $N' \succeq_{cK} \{\!\!\{N\}\!\!\}$).

2. $C[\cdot] \mapsto C'[\cdot]$ and $\bar{N} \equiv C'[\llbracket N \rrbracket]$. This case is trivial.
3. $\llbracket N \rrbracket \xrightarrow{\triangleright l} N'$, $C[\cdot] \equiv C[\cdot \parallel l :: \mathbf{nil}]$ and $\bar{N} \equiv C[N']$. Because of Lemma 6.4.2.6, we have two possible sub-cases:
 - (a) $N \xrightarrow{\triangleright l} N''$ and $N' \succeq_{cK} \llbracket N'' \rrbracket$. In this case it is easily to conclude.
 - (b) $N \equiv \mathcal{D}[l' :: \mathbf{in}(T)@l.P]$, for $l \notin \text{bn}(\mathcal{D}[\cdot]) \cup \text{fn}(N)$, and $N' \equiv \llbracket \mathcal{D} \rrbracket[l' :: \mathbf{nil} \parallel l :: \mathbf{in}(T).\mathbf{eval}(\llbracket P \rrbracket_{l'})@l']$. Now, $C[\llbracket N \rrbracket] \mapsto C[\llbracket \mathcal{D} \rrbracket[l' :: \mathbf{nil} \parallel l :: \mathbf{in}(T).\mathbf{eval}(\llbracket P \rrbracket_{l'})@l']] \triangleq \bar{N} \succeq_{cK} C[\llbracket \mathcal{D}[l' :: \mathbf{nil} \parallel l :: \mathbf{in}(T).\mathbf{eval}(\llbracket P \rrbracket_{l'})@l'] \rrbracket] \triangleq C[\llbracket N \parallel l :: \mathbf{nil} \rrbracket]$ (the last inequality holds by Proposition 6.4.1.3). Now, since $C[N] \equiv C[N \parallel l :: \mathbf{nil}]$, we have that $(C[N], \bar{N}) \in \mathfrak{X}$ up-to \preceq_{cK} , as required.
4. $\llbracket N \rrbracket \xrightarrow{\mathbf{nil} @ l} N'$, $C[\cdot] \equiv C'[\cdot \parallel H]$, $H \xrightarrow{\triangleright l} H'$ and $\bar{N} \equiv C'[N' \parallel L']$. This case relies on Lemma 6.4.2.4 and is simple.
5. $\llbracket N \rrbracket \xrightarrow{l' \triangleleft l} N'$, $C[\cdot] \equiv C'[\cdot \parallel l :: \langle l' \rangle]$ and $\bar{N} \equiv C'[N']$. The proof relies on Lemma 6.4.2.5 and is simple.
6. $\llbracket N \rrbracket \xrightarrow{(\widetilde{v}l) \langle l' \rangle @ l} N'$, $C[\cdot] \equiv C'[\cdot \parallel H]$, $H \xrightarrow{l' \triangleleft l} H'$ and $\bar{N} \equiv C'[(\widetilde{v}l)(N' \parallel H')]$. This case relies on Lemma 6.4.2.4 and is simple. ■

Corollary 6.4.5 (Semantical Equivalence w.r.t. \cong_{cK}) *Let N be a cKLAIM net. Then, $N \cong_{cK} \llbracket N \rrbracket$.*

Proof: By Propositions 6.4.1.4 and 6.1.2, Theorem 6.4.4 and by transitivity of \cong_{cK} . ■

6.5 A Comparison with the π_a -calculus

Finally, we want to assess the overall expressive power of our KLAIM-based languages by comparing (some of) them with a proper variant of the π -calculus. The variant we choose is the asynchronous π -calculus taken from [19] plus the name matching construct; we call the resulting formalism π_a -calculus. Its syntax is

$$p ::= \mathbf{0} \mid \bar{a}b \mid a(b).p \mid p_1|p_2 \mid (va)p \mid [a = b]p \mid !p$$

We want to remark that we consider here only the monadic version of the calculus; using a polyadic π_a -calculus does not change the results that we shall prove in this section at all; we would have just used μKLAIM in place of cKLAIM for the comparison.

The operational semantics of the calculus is given in Table 6.5, by following [5]. On top of this LTS, barbed equivalence is defined as follows (see also [5]).

$\bar{a}b \rightarrow \mathbf{0}$	$a(b).p \xrightarrow{ac} p[c/b]$	$\frac{p \xrightarrow{\mu} p'}{[a = a]p \xrightarrow{\mu} p'}$
$\frac{p \xrightarrow{\mu} p'}{!p \xrightarrow{\mu} !p \mid p'}$	$\frac{p \xrightarrow{\mu} p' \quad b \notin \text{fn}(\mu)}{(vb)p \xrightarrow{\mu} (vb)p'}$	$\frac{p \xrightarrow{\bar{a}b} p' \quad a \neq b}{(vb)p \xrightarrow{\bar{a}(b)} p'}$
$\frac{p =_{\alpha} p' \xrightarrow{\mu} q' =_{\alpha} q}{p \xrightarrow{\mu} q}$		$\frac{p \xrightarrow{\bar{a}b} p' \quad q \xrightarrow{ab} q'}{p \mid q \xrightarrow{\tau} p' \mid q'} (*)$
$\frac{p \xrightarrow{\bar{a}(b)} p' \quad q \xrightarrow{ab} q' \quad b \notin \text{fn}(q)}{p \mid q \xrightarrow{\tau} (vb)(p' \mid q')} (*)$		$\frac{p \xrightarrow{\mu} p' \quad \text{bn}(\mu) \cap \text{fn}(q) = \emptyset}{p \mid q \xrightarrow{\mu} p' \mid q} (*)$

and the obvious symmetric versions of the rules marked with (*)

Table 6.5: A LTS for the π_a -calculus

Definition 6.5.1 (Asynchronous Barbed Equivalence) Asynchronous barbed equivalence, \cong_{π_a} , is the largest symmetric relation between π_a -calculus processes such that $p \cong_{\pi_a} q$ implies that

1. whenever $p \Downarrow \bar{a}$, it holds that $q \Downarrow \bar{a}$,
where $p \Downarrow \bar{a} \triangleq (\exists b. p \xrightarrow{\bar{a}b} \vee p \xrightarrow{\bar{a}(b)})$ and $p \Downarrow \bar{a} \triangleq (p \Rightarrow \downarrow \bar{a})$
2. whenever $p \xrightarrow{\tau} p'$, it holds that $q \Rightarrow q'$ and $p' \cong_{\pi_a} q'$
3. for all names \tilde{n} and for all π_a -calculus process r , it holds that $(\tilde{v}\tilde{n})p \cong_{\pi_a} (\tilde{v}\tilde{n})q$ and $p \mid r \cong_{\pi_a} q \mid r$.

6.5.1 Encoding the π_a -calculus in cKLAIM

We now provide an encoding of the π_a -calculus in cKLAIM; it is given in Table 6.6. Like in the previous section, we need a normalisation function between cKLAIM nets that makes the encoding prompt. It is defined as follows:

$$\begin{aligned}
nrm_L((\nu l)N) &\triangleq nrm_{L \cup \{l\}}(N) \\
nrm_L(N_1 \parallel N_2) &\triangleq nrm_L(N_1) \parallel nrm_L(N_2) \\
nrm_L(l :: \langle \cdot \rangle) &\triangleq l :: \langle \cdot \rangle \\
nrm_L(l :: C_1 \mid C_2) &\triangleq nrm_L(l :: C_1) \parallel nrm_L(l :: C_2) \\
nrm_L(l :: P) &\triangleq \begin{cases} l :: P' \parallel l' :: \langle l'' \rangle & \text{if } P = \mathbf{out}(l'')@l'.P' \text{ and } l' \in L \\ (\nu l')(nrm_{L \cup \{l'\}}(l :: P')) & \text{if } P = \mathbf{new}(l').P' \\ l :: P & \text{if } P \neq _ \mid _ \text{ and no previous case holds} \end{cases}
\end{aligned}$$

Top-level Encoding:	
$\llbracket p \rrbracket_L \triangleq \text{proc} :: \llbracket p \rrbracket \parallel \prod_{n \in L} n :: \mathbf{nil}$	if $fn(p) \subseteq L$ and proc is a reserved name
Encoding π_a-calculus processes:	
$\llbracket 0 \rrbracket \triangleq \mathbf{nil}$	$\llbracket (va)p \rrbracket \triangleq \mathbf{new}(a).\llbracket p \rrbracket$
$\llbracket \bar{a}b \rrbracket \triangleq \mathbf{out}(b)@a.\mathbf{nil}$	$\llbracket a(b).p \rrbracket \triangleq \mathbf{in}(!b)@a.\llbracket p \rrbracket$
$\llbracket p_1 p_2 \rrbracket \triangleq \llbracket p_1 \rrbracket \parallel \llbracket p_2 \rrbracket$	$\llbracket !p \rrbracket \triangleq \mathbf{rec} X.(\llbracket p \rrbracket X)$
$\llbracket [a = b]p \rrbracket \triangleq \mathbf{new}(l).\mathbf{out}(a)@l.\mathbf{in}(b)@l.\llbracket p \rrbracket$	

Table 6.6: Encoding π_a -calculus in cKLAIM

Essentially, the normalisation replaces all actions **new** with the net resulting from the creation of the new nodes and all actions **out** over existing localities with the net containing the datum produced by the action. When a net is the encoding of a π_a -calculus process, the continuation of each action **out** is **nil** and function nrm does not need to be iterated on it.

For the sake of readability, we write $nrm_L(\llbracket p \rrbracket_L)$ as $\llbracket \llbracket p \rrbracket \rrbracket_L$. Some simple but crucial properties of $nrm_L(\cdot)$ are given in the following proposition, whose proof is simple.

Proposition 6.5.2 *Let P be a cKLAIM process and p be a π_a -calculus process. Then*

1. if $l \neq l'$ then $nrm_L((vl')(l :: P)) = nrm_L(l :: \mathbf{new}(l').P)$
2. $\llbracket p \rrbracket_L \parallel l :: \mathbf{nil} \succeq_{cK} \llbracket p \rrbracket_{L \cup \{l\}}$
3. $\llbracket p \rrbracket_L \succeq_{lcK} \llbracket \llbracket p \rrbracket \rrbracket_L$

We now prove a tight correspondence between π_a -calculus processes and their encodings. We start with a correspondence between the labelled semantics of the two calculi and then give their operational correspondence.

Lemma 6.5.3 *Let p be a π_a -calculus process and $fn(p) \subseteq L$. Then*

1. if $p \xrightarrow{\bar{a}b} p'$ then $\llbracket p \rrbracket_L \xrightarrow{\langle b \rangle @ a} \llbracket p' \rrbracket_L$
2. if $p \xrightarrow{\bar{a}(b)} p'$ then $\llbracket p \rrbracket_L \xrightarrow{(vb) \langle b \rangle @ a} \llbracket p' \rrbracket_{L \cup \{b\}}$
3. if $p \xrightarrow{ab} p'$ then $\llbracket p \rrbracket_L \xrightarrow{b \triangleleft a} N$ and $\llbracket p' \rrbracket_{L \cup \{b\}} \equiv N \parallel b :: \mathbf{nil}$
4. if $\llbracket p \rrbracket_L \xrightarrow{\langle b \rangle @ a} N$ then $p \xrightarrow{\bar{a}b} p'$ and $N \equiv \llbracket p' \rrbracket_L$
5. if $\llbracket p \rrbracket_L \xrightarrow{(vb) \langle b \rangle @ a} N$ then $p \xrightarrow{\bar{a}(b)} p'$ and $\llbracket p' \rrbracket_{L \cup \{b\}} \equiv N \parallel b :: \mathbf{nil}$
6. if $\llbracket p \rrbracket_L \xrightarrow{b \triangleleft a} N$ then $p \xrightarrow{ab} p'$ and $\llbracket p' \rrbracket_{L \cup \{b\}} \preceq_{cK} N \parallel b :: \mathbf{nil}$

Proof: By induction on the length of the inferences. ■

Lemma 6.5.4 (Operational Correspondence) *Let p be a π_a -calculus process and $fn(p) \subseteq L$. Then*

1. $p \xrightarrow{\tau} p'$ implies that $\llbracket p \rrbracket_L \Rightarrow \llbracket p' \rrbracket_L$
2. $\llbracket p \rrbracket_L \xrightarrow{\tau} N$ implies that $p \xrightarrow{\tau} p'$ and $N \succeq_{cK} \llbracket p' \rrbracket_L$

Proof: Both the claims are proved by induction on the inference occurring in the premise. The first statement is quite simple. We give the base cases for the second statement. We want to remark that, thanks to the normalisation procedure, the only possible base case is when using rule (LTS-COMM). Thus, $\llbracket p \rrbracket_L \triangleq N_1 \parallel N_2 \xrightarrow{\tau} N'_1 \parallel N'_2 \triangleq N$, because $N_1 \xrightarrow{b \triangleleft a} N'_1$ and $N_2 \xrightarrow{\langle b \rangle @ a} N'_2$. By definition, it must be that $N_i \triangleq \llbracket p_i \rrbracket_L$, for $i = 1, 2$; moreover, $\{a, b\} \subseteq fn(N_2) \subseteq L$. By Lemma 6.5.3.6 and .4, we have that $p_1 \xrightarrow{ab} p'_1$ and $\llbracket p'_1 \rrbracket_L \lesssim_{cK} N'_1 \parallel b :: \mathbf{nil}$, and $p_2 \xrightarrow{\bar{a}b} p'_2$ and $N'_2 \equiv \llbracket p'_2 \rrbracket_L$. Thus, $p \triangleq p_1 | p_2 \xrightarrow{\tau} p'_1 | p'_2 \triangleq p'$. Moreover, $N \equiv N'_1 \parallel b :: \mathbf{nil} \parallel N'_2 \succeq_{cK} \llbracket p'_1 \rrbracket_L \parallel \llbracket p'_2 \rrbracket_L \triangleq \llbracket p' \rrbracket_L$, as required. ■

We now prove that full abstraction can be obtained when considering only the following subset of cKLAIM contexts, called *translated*:

$$C[\cdot] ::= [\cdot] \quad | \quad C[\cdot] \parallel \llbracket p \rrbracket_L \quad | \quad (\nu l)C[\cdot]$$

Basically, we only permit parallel components resulting from the encoding of π_a -calculus processes. This ensures that each free name occurring in any parallel component is also the address of a node in the component itself; this is crucial to prove full abstraction.

Theorem 6.5.5 *Let $fn(p, q) \subseteq L$. Then $p \cong_{\pi_a} q$ if and only if $\llbracket p \rrbracket_L \cong_{cK}^{\text{tr}} \llbracket q \rrbracket_L$.*

Proof: We start with proving that $\llbracket p \rrbracket_L \cong_{cK}^{\text{tr}} \llbracket q \rrbracket_L$ implies $p \cong_{\pi_a} q$. Let $\mathfrak{R} \triangleq \{(p, q) : \llbracket p \rrbracket_L \cong_{cK}^{\text{tr}} \llbracket q \rrbracket_L\}$; we prove that $\mathfrak{R} \subseteq \cong_{\pi_a}$. Let $p \mathfrak{R} q$. For reduction closure, we take $p \xrightarrow{\tau} p'$; by Lemma 6.5.4.1 it holds that $\llbracket p \rrbracket_L \Rightarrow \llbracket p' \rrbracket_L$. Thus, $\llbracket q \rrbracket_L \Rightarrow N$ and $\llbracket p' \rrbracket_L \cong_{cK}^{\text{tr}} N$. Then, by using Lemma 6.5.4.2, it can be easily verified that $q \Rightarrow q'$ and $\llbracket q' \rrbracket_L \lesssim_{cK} N$. This suffices to conclude that $p' \mathfrak{R} q'$ since, by the fact that $\lesssim_{cK} \subseteq \cong_{cK} \subseteq \cong_{cK}^{\text{tr}}$ and by transitivity of \cong_{cK}^{tr} , it holds that $\llbracket p' \rrbracket_L \cong_{cK}^{\text{tr}} \llbracket q' \rrbracket_L$ (notice that, as it is standard in the π -calculus, $fn(p') \subseteq fn(p)$ and, thus, $fn(p') \subseteq L$ – and similarly for q and q').

We now consider barb preservation; let $p \downarrow \bar{a}$ because $p \xrightarrow{\bar{a}b}$. By Lemma 6.5.3.1 it holds that $\llbracket p \rrbracket_L \xrightarrow{b @ a}$; thus, $\llbracket q \rrbracket_L \xrightarrow{b @ a}$. Hence, by Lemma 6.5.3.4 and reduction closure (just proved), it holds that $q \xrightarrow{\bar{a}b}$; thus, by definition, $q \downarrow \bar{a}$. The case for $p \downarrow \bar{a}$ because $p \xrightarrow{\bar{a}(b)}$ is similar, but relies on Lemma 6.5.3.2 and .5.

Finally, we have to prove closure under parallel composition and restriction. Let us examine the two conditions separately.

- We want to prove that $(\widehat{v\bar{n}})p \mathfrak{R} (\widehat{v\bar{n}})q$ by knowing that $(\widehat{v\bar{n}})(\llbracket p \rrbracket_L) \cong_{cK}^{tr} (\widehat{v\bar{n}})(\llbracket q \rrbracket_L)$. By definition, we have that $(\widehat{v\bar{n}})(\llbracket \cdot \rrbracket_L) \triangleq \llbracket (\widehat{v\bar{n}}) \cdot \rrbracket_{L-\{\bar{n}\}}$ and $fn((\widehat{v\bar{n}}) \cdot) \triangleq fn(\cdot) - \{\bar{n}\}$. Thus, $fn((\widehat{v\bar{n}})p, (\widehat{v\bar{n}})q) \subseteq L - \{\bar{n}\}$ and hence $\llbracket (\widehat{v\bar{n}})p \rrbracket_{L-\{\bar{n}\}} \cong_{cK}^{tr} \llbracket (\widehat{v\bar{n}})q \rrbracket_{L-\{\bar{n}\}}$. By definition of \mathfrak{R} , this suffices to conclude.
- We want to prove that $p|r \mathfrak{R} q|r$ by knowing that $\llbracket p \rrbracket_L \parallel \llbracket r \rrbracket_{L'} \cong_{cK}^{tr} \llbracket q \rrbracket_L \parallel \llbracket r \rrbracket_{L'}$. By definition of $\llbracket \cdot \rrbracket$, and by Proposition 6.5.2.2, it holds that $\llbracket \cdot \rrbracket_L \parallel \llbracket r \rrbracket_{L'} \succeq_{cK} \llbracket \cdot \rrbracket_{L \cup L'} \parallel \llbracket r \rrbracket_{L \cup L'} \triangleq \llbracket \cdot \parallel r \rrbracket_{L \cup L'}$. Thus, $\llbracket p \parallel r \rrbracket_{L \cup L'} \cong_{cK}^{tr} \llbracket q \parallel r \rrbracket_{L \cup L'}$ and $fn(p|r, q|r) = fn(p, q) \cup fn(r) \subseteq L \cup L'$. This suffices to conclude.

We are left with proving the converse, i.e. $p \approx_{\pi_a} q$ implies that $\llbracket p \rrbracket_L \cong_{cK}^{tr} \llbracket q \rrbracket_L$. Let $\mathfrak{R} \triangleq \{(\llbracket p \rrbracket_L, \llbracket q \rrbracket_L) : p \approx_{\pi_a} q\}$; we prove that \mathfrak{R} is a barbed congruence, up-to \preceq_{cK} . For reduction closure, we let $\llbracket p \rrbracket_L \xrightarrow{\tau} N$; by Lemma 6.5.4.2 it holds that $p \xrightarrow{\tau} p'$ and $N \succeq_{cK} \llbracket p' \rrbracket_L$. Then, $q \Rightarrow q'$ and $p' \approx_{\pi_a} q'$. By Lemma 6.5.4.1, we know that $\llbracket q \rrbracket_L \Rightarrow \llbracket q' \rrbracket_L$; this suffices to conclude up-to \preceq_{cK} . Barb preservation can be proved easily. By Proposition 5.2.3.2 (or .3), $\llbracket p \rrbracket_L \downarrow a$ implies that $\llbracket p \rrbracket_L \xrightarrow{(v\bar{b}) \langle b \rangle @ a}$; by Lemma 6.5.3.4 (or .5) and by definition of barbs in the π_a -calculus, this implies that $p \downarrow \bar{a}$. Then, $q \downarrow \bar{a}$; by using Lemma 6.5.3.1 (or .2) and Lemma 6.5.4.1, we obtain the desired $\llbracket q \rrbracket_L \downarrow a$.

To conclude, we have to prove that, for every translated context $C[\cdot]$, it holds that $C[\llbracket p \rrbracket_L] \mathfrak{R} C[\llbracket q \rrbracket_L]$. The key observation is that, by definition of translated context, it holds that $C[\cdot] \equiv (\widehat{v\bar{n}})([\cdot] \parallel \llbracket r \rrbracket_{L'})$. Moreover, by hypothesis, we know that $(\widehat{v\bar{n}})(p|r) \approx_{\pi_a} (\widehat{v\bar{n}})(q|r)$. Hence,

$$\begin{aligned}
C[\llbracket \cdot \rrbracket_L] &\equiv (\widehat{v\bar{n}})(\llbracket \cdot \rrbracket_L \parallel \llbracket r \rrbracket_{L'}) \\
&\succeq_{cK} (\widehat{v\bar{n}})(\llbracket \cdot \rrbracket_L \parallel \llbracket r \rrbracket_{L'}) && \text{by Prop. 6.5.2.3} \\
&\succeq_{cK} (\widehat{v\bar{n}})(\llbracket \cdot \rrbracket_{L \cup L'} \parallel \llbracket r \rrbracket_{L \cup L'}) && \text{by Prop. 6.5.2.2} \\
&\triangleq (\widehat{v\bar{n}})(nrm_{L \cup L'}(\llbracket \cdot \parallel r \rrbracket_{L \cup L'})) \\
&\triangleq nrm_{L''}((\widehat{v\bar{n}})(\llbracket \cdot \parallel r \rrbracket_{L \cup L'})) && \text{for } L'' \triangleq (L \cup L') - \{\bar{n}\} \\
&\triangleq nrm_{L''}((\widehat{v\bar{n}})(\text{proc} :: \llbracket \cdot \parallel r \rrbracket \parallel \prod_{l' \in L \cup L'} l' :: \mathbf{nil})) \\
&\equiv nrm_{L''}((\widehat{v\bar{n}})(\text{proc} :: \llbracket \cdot \parallel r \rrbracket)) \parallel \prod_{l' \in L''} l' :: \mathbf{nil} \\
&= nrm_{L''}(\text{proc} :: \llbracket (\widehat{v\bar{n}})(\cdot \parallel r) \rrbracket) \parallel \prod_{l' \in L''} l' :: \mathbf{nil} && \text{by Prop. 6.5.2.1} \\
&\equiv \llbracket (\widehat{v\bar{n}})(\cdot \parallel r) \rrbracket_{L''}
\end{aligned}$$

Notice that, if $fn(\cdot) \subseteq L$ and $fn(r) \subseteq L'$ (these hold by definition of the encoding), then $fn((\widehat{v\bar{n}})(\cdot \parallel r)) \subseteq (L \cup L') - \{\bar{n}\} \triangleq L''$. Thus, $C[\llbracket p \rrbracket_L] \succeq_{cK} \llbracket (\widehat{v\bar{n}})(p|r) \rrbracket_{L''} \mathfrak{R} \llbracket (\widehat{v\bar{n}})(q|r) \rrbracket_{L''} \preceq_{cK} C[\llbracket q \rrbracket_L]$. This suffices to conclude, up-to \preceq_{cK} . \blacksquare

Corollary 6.5.6 (Full Abstraction w.r.t. Translated Barbed Equivalence) *Let $fn(p, q) \subseteq L$. Then $p \approx_{\pi_a} q$ if and only if $\llbracket p \rrbracket_L \cong_{cK}^{tr} \llbracket q \rrbracket_L$.*

Proof: Trivial, by Theorem 6.5.5, Proposition 6.5.2.3 and by observing that $\lesssim_{cK} \subseteq \cong_{cK} \subseteq \cong_{cK}^{\text{tr}}$. ■

Remark 6.5.7: On full abstraction w.r.t. barbed equivalence. We have already said that translated full abstraction seems us the best possible result for the encoding of Table 6.6. Indeed, there is a key design issue that breaks full abstraction: in π -calculus, knowing a name implies that communications can be performed upon a channel with that name and these actions succeed whenever a parallel component performs a complementary action. This is not the case in KLAIM (and in the calculi derived from it). Indeed, it is *not* necessarily the case that each free name is associated to a locality (while each name in a π -calculus process is associated to a channel). This aspect can break full abstraction: e.g., consider the following π_a -calculus equivalence

$$p \triangleq a(x).(\bar{x} \mid \bar{x}b) \cong_{\pi_a} a(x).((\bar{x} \mid \bar{x}b) \oplus \bar{b}) \triangleq q$$

where \oplus denotes internal choice. However,

$$\llbracket p \rrbracket_L \not\equiv_{cK} \llbracket q \rrbracket_L$$

Indeed, $\llbracket q \rrbracket_L$ can produce a datum at node b , while $\llbracket p \rrbracket_L$ cannot: if the name received in the input (that replaces x) is not a node of the net, the encoding of the output over x will never produce a datum. Thus, the input from x is blocked and the following output on b will never produce a datum.

We think that no ‘reasonable’ encoding of π_a -calculus in cKLAIM (nor any other calculus derived from KLAIM) can be given: checking existence of nodes before firing an output is a too low-level feature that cannot be implemented in such an abstract setting as the π -calculus. There are two ways in which we can recover full abstraction.

1. We can make cKLAIM higher-level: a simple way to do this is to add the following structural rule to those given in Table 2.3

$$l :: \mathbf{nil} \equiv \mathbf{0}$$

In this way, we recover the π -calculus’ philosophy that each name is *always* associated to a communication medium (up-to \equiv).

Another possibility is to consider a typed language, where types ensure that, if a locality name is eventually used as target of an operation, then a node whose address is that name is present in the net. This strongly resembles $D\pi$ ’s framework [90].

2. We can make the π_a -calculus lower-level: some names are channels, while the other ones are just communicable objects. This can be formalised by structuring the syntax of the π_a -calculus as follows:

$$\begin{array}{l} \text{Systems } S ::= \exists a \mid (va)S \mid S_1 \mid S_2 \mid p \\ \text{Processes } p ::= \dots \end{array}$$

Encoding Nets:	
$\langle \mathbf{0} \rangle \triangleq !\overline{e\bar{x}}\langle \rangle$	$\langle N_1 \parallel N_2 \rangle \triangleq \langle N_1 \rangle \mid \langle N_2 \rangle$
$\langle (\nu l)N \rangle \triangleq (\nu l)(\langle N \rangle \mid !\overline{e\bar{x}}l)$	$\langle l :: C \rangle \triangleq \langle C \rangle_l \mid !\overline{e\bar{x}}l$
Encoding Processes:	
$\langle \mathbf{nil} \rangle_u \triangleq \mathbf{0}$	$\langle \langle l \rangle \rangle_u \triangleq \bar{u}l$
$\langle X \rangle_u \triangleq X$	$\langle \mathbf{rec} X.P \rangle_u \triangleq \mathbf{rec} X.\langle P \rangle_u$
$\langle C_1 \mid C_2 \rangle_u \triangleq \langle C_1 \rangle_u \mid \langle C_2 \rangle_u$	$\langle \mathbf{new}(l).P \rangle_u \triangleq (\nu l)(\langle P \rangle_u \mid !\overline{e\bar{x}}l)$
$\langle \mathbf{out}(u').P \rangle_u \triangleq \bar{u}u \mid \langle P \rangle_u$	$\langle \mathbf{in}(!x).P \rangle_u \triangleq u(x).\langle P \rangle_u$
$\langle \mathbf{in}(u').P \rangle_u \triangleq \mathbf{rec} X.u(x).(vc)(\bar{c} \mid [x = \bar{u}]c.\langle P \rangle_u \mid c.(\bar{u}x \mid X))$	(*)
$\langle \mathbf{eval}(Q)@u'.P \rangle_u \triangleq \mathbf{rec} X.ex(x).(vc)(\bar{c} \mid [x = \bar{u}]c.(\langle P \rangle_u \mid \langle Q \rangle_{u'}) \mid c.X)$	(*)
(*) for x, X fresh	

Table 6.7: Encoding lcKLAIM in π_a -calculus

where the particle $\exists a$ implements the presence of a channel with name a . The operational semantics of Table 6.5 must be then modified by following the lines of the LTS in Table 3.8 (by adding a check of existence of a channel before firing an output action). We call π_a^\exists this lower-level calculus.

6.5.2 Encoding lcKLAIM in the π_a -calculus

We now present an encoding of the simplest KLAIM -based calculus, namely lcKLAIM , in the π_a -calculus. The encoding is somehow inspired from the encoding of KLAIM in μKLAIM (for the handling of names) and of μKLAIM in cKLAIM (for the encoding of the name matching construct of lcKLAIM).

We can follow the correspondence between channels and localities that we pointed out in Section 6.5.1 and translate each locality to a channel. Output actions performed at l , as well as data located at l , can be translated to output particles of the π_a -calculus $\bar{l} _$. Similarly, input actions performed at l can be translated to input prefixes of the π_a -calculus $l(x) _$. Finally, any action $\mathbf{new}(l')$ is translated to a restriction $(\nu l')$. Thus, the correspondence between the two calculi is quite straightforward up to now.

A first feature that distinguish lcKLAIM from π_a -calculus is the communication paradigm and, mainly, the name matching of lcKLAIM (that happens while retrieving a datum). This issue can be encoded quite easily, if we accept divergence: process $\mathbf{in}(l').P$ running at l can be translated into a process that first retrieves a datum at l and then checks if it is l' ; if the check succeeds, the process continues, otherwise it places back the accessed datum and looks for another one.

A second feature that distinguish lcKLAIM from π_a -calculus is the allocation of processes and their movements, together with the check of locality existence before

migration. Process distribution is relevant in LCKLAIM to establish where actions **out** and **in** have to take place. Thus, we can define a parameterised encoding for processes, $\llbracket P \rrbracket_u$, where u is the locality where P runs. Then, if P is of the form **out**(u'). Q , we translate it to $\bar{c}u' \mid \llbracket Q \rrbracket_u$, while, if P is of the form **in**($!x$). Q , we translate it to $u(x).\llbracket Q \rrbracket_u$. A process P of the form **eval**(Q)@ u' . R running at u is translated to the parallel composition of $\llbracket R \rrbracket_u$ and $\llbracket Q \rrbracket_{u'}$, if existence of locality u' is ascertained.

The last feature we have to model is the distinction between names that are addresses of network nodes and raw names. The former ones can be then used as target of remote operations (in the case of LCKLAIM only actions **eval**), while the latter ones cannot. By using a π -calculus terminology, only the first ones are *Names* (we intentionally used the capital letter), while the latter ones are just *values*. However, the status of a name (without the capital letter) can change according to the context: a Name will always remain such in any context, while a value l can become a Name if the context provides a node with address l .

To deal with this sophisticated feature (that, as we have already discussed in Remark 6.5.7, creates a relevant gap between π_a -calculus and KLAIM -based calculi), we use a reserved channel **ex** to record existence of localities. Thus, if l is a Name in the LCKLAIM net considered for translation, then channel **ex** will repeatedly offer l in the encoded net, i.e. the encoding will contain a process of the form

$$! \bar{\text{ex}} l \triangleq (vc)(\bar{c} l \mid !c(x)(\bar{c} x \mid \text{ex } x))$$

The encoding is reported in Table 6.7. There, we also assume the possibility of writing π_a -calculus processes with recursion – that can be implemented through replication, as usual – and we write \bar{c} and c to mean output and input of dummy data. In the translation of actions **in**(l) and **eval**, the fresh restricted channel c is used to implement a form of *internal choice*. In both cases, the first addendum can evolve only if the name matching succeeds. On the other hand, the second addendum can always be executed: this fact introduces divergence in the encoding. Notice, however, that exactly one of the two addendum can evolve. Finally, like in the encoding of KLAIM in μKLAIM , the fact that **ex** always provides data is necessary to obtain a fully abstraction result w.r.t. translated contexts. Again, translated contexts do not have a full discriminating power over this channel.

The proof of soundness somehow follows proofs already given in the chapter; we only sketch the main steps and leave the details to the interested reader. First, the translated barbed expansion in the π_a -calculus, written $\lesssim_{\pi_a}^{\text{tr}}$, can be defined by following Definition 6.1.1. By following Proposition 6.1.2, it can be proved that $\lesssim_{\pi_a}^{\text{tr}} \subset \cong_{\pi_a}^{\text{tr}}$ and, by following Lemma 6.1.5, that translated barbed congruence up to $\lesssim_{\pi_a}^{\text{tr}}$ is contained in $\cong_{\pi_a}^{\text{tr}}$. Then, we can prove that each LCKLAIM reduction is preserved by its encoding.

Lemma 6.5.8 *If $N \mapsto N'$, then $\llbracket N \rrbracket \Rightarrow \gtrsim_{\pi_a}^{\text{tr}} \llbracket N' \rrbracket$.*

Now, since the encoding $\llbracket \cdot \rrbracket$ is divergent, we can follow the ideas of Section 6.3 and define partial reducts and partial states. Notice that, since divergence can orig-

inate both from the encoding of name matching and of migrations, we have two possible cases for partial reducts.

Definition 6.5.9

1. A π_a -calculus process p is a partial reduct of a LCKLAIM net N whenever

- $N \equiv l :: \mathbf{in}(l').P \mid \langle l' \rangle$ and
 $p \cong_{\pi_a}^{\text{tr}} (\nu c)(\bar{c} \mid [x = \mathcal{T}]c.(\mathbb{P})_u \mid c.(\bar{l}x \mid (\mathbf{in}(T).P)_l)) \mid !\bar{\mathbf{ex}}l$, or
- $N \equiv l :: \mathbf{eval}(Q)@l'.P \parallel l' :: \mathbf{nil}$ and
 $p \cong_{\pi_a}^{\text{tr}} (\nu c)(\bar{c} \mid [x = \mathcal{T}]c.((\mathbb{P})_l \mid (\mathbb{Q})_r) \mid c.(\mathbf{eval}(Q)@l'.P)_l) \mid !\bar{\mathbf{ex}}l \mid !\bar{\mathbf{ex}}l'$.

2. A π_a -calculus process p is a partial state of a LCKLAIM net N whenever $N \equiv (\nu \bar{l})(N_1 \parallel \dots \parallel N_n \parallel \bar{N})$, $p \equiv_{\pi} (\nu \bar{l})(p_1 \parallel \dots \parallel p_n \parallel (\bar{N}))$ and for all i it holds that p_i is a partial reduct of N_i (where \equiv_{π} is Milner's structural equivalence, see [112]).

The pleasant property of μKLAIM 's partial reducts is here stronger.

Lemma 6.5.10 *Let p be a partial reduct of N . Then,*

- $N \equiv l :: \mathbf{in}(l').P \mid \langle l' \rangle$ and $p \xrightarrow{\tau} p'$ imply that either $p' \gtrsim_{\pi_a}^{\text{tr}} (\mathbb{N})$, or $p' \gtrsim_{\pi_a}^{\text{tr}} (\mathbb{P})_l \mid !\bar{\mathbf{ex}}l$.
- $N \equiv l :: \mathbf{eval}(Q).P \parallel l' :: \mathbf{nil}$ and $p \xrightarrow{\tau} p'$ imply that either $p' \gtrsim_{\pi_a}^{\text{tr}} (\mathbb{N})$, or $p' \gtrsim_{\pi_a}^{\text{tr}} (\mathbb{P})_l \mid (\mathbb{Q})_r \mid !\bar{\mathbf{ex}}l \mid !\bar{\mathbf{ex}}l'$.

We can now state the reflection of reduction steps.

Lemma 6.5.11 *If $\langle N \rangle \xrightarrow{\tau} p$, then either $N \xrightarrow{\tau} N'$ and $p \gtrsim_{\pi_a}^{\text{tr}} \langle N' \rangle$, or p is a partial state of N .*

Finally, it is easy to see that the encoding faithfully translates the barbs; it only adds new barbs on \mathbf{ex} but no translated context can fully observe them. Hence, the proof of the following concluding theorem can be carried on easily.

Theorem 6.5.12 (Full Abstraction w.r.t. Translated Barbed Congruence)

$N \cong_{l_c K} M$ if and only if $(\mathbb{N}) \cong_{\pi_a}^{\text{tr}} (\mathbb{M})$.

Proof: For the ‘if’ direction, it suffices to prove that relation

$$\begin{aligned} \mathfrak{R} &\triangleq \{(N, M) : (\mathbb{N}) \cong_{\pi_a}^{\text{tr}} (\mathbb{M})\} \\ &\cup \{(N, M) : \exists \bar{p}. (\mathbb{N}) \stackrel{\text{tr}}{\gtrsim}_{\pi_a} \bar{p} \wedge \bar{p} \text{ partial state of } M\} \\ &\cup \{(N, M) : \exists \bar{p}. \langle M \rangle \stackrel{\text{tr}}{\gtrsim}_{\pi_a} \bar{p} \wedge \bar{p} \text{ partial state of } N\} \end{aligned}$$

is barb preserving, reduction closed (up-to \lesssim_{lcK}) and closed under translated contexts (again, up-to \lesssim_{lcK}). For the ‘only if’ direction, it suffices to prove that relation

$$\begin{aligned} \mathfrak{R} \triangleq & \bigcup_{N \approx_{lcK} M} \{(\llbracket N \rrbracket, \llbracket M \rrbracket)\} \\ & \cup \{(\llbracket N \rrbracket, p) : p \text{ partial state of } M\} \\ & \cup \{(p, \llbracket M \rrbracket) : p \text{ partial state of } N\} \end{aligned}$$

is barb preserving, reduction closed (up-to $\lesssim_{\pi_a}^{\text{tr}}$) and closed under translated contexts (again, up-to $\lesssim_{\pi_a}^{\text{tr}}$). ■

6.6 Concluding Assessment and Related Work

The main results of this chapter are summarised in Table 6.8. There, a labelled arrow between two calculi, $\mathcal{X} \xrightarrow{\mathcal{P}} \mathcal{Y}$, means that the language \mathcal{X} can be encoded in the language \mathcal{Y} and the encoding enjoys property \mathcal{P} . The arrow is dotted if the encoding can introduce divergence, i.e. infinite sequences of reductions that in the source term were not present.

According to Palamidessi [122], each ‘reasonable’ encoding $enc(\cdot)$ should enjoy at the very least the following properties:

1. *homomorphic w.r.t. the parallel operator*, i.e. $enc(N \parallel M) = enc(N) \parallel enc(M)$ (this is needed in order to maintain the degree of parallelism during the translation);
2. *preserving renaming*, i.e. for every permutation of names σ in the source language there exists a permutation of names θ in the target language such that $enc(P\sigma) = (enc(P))\theta$ (this ensures that the translation does not depend on the names involved);
3. *preserving the basic observables*, i.e. it has to preserve the visible behaviours of the encoded terms;
4. *preserving termination*, i.e. it has to turn each terminating term in a terminating term.

All the encodings we have presented in this chapter enjoy properties 2. and 3. . About property 4., it is not enjoyed by the encodings of μKLAIM in cKLAIM and of lcKLAIM into the π_a -calculus. This is related to the fact that, in our opinion, the form of name matching present in the KLAIM -based calculi (that, in turn, has been inherited by LINDA [74]) is very powerful: it allows to perform boolean tests on names while retrieving them. A similar feature led to the separation result of [122] between the synchronous and the asynchronous π -calculus. About property 1., it is not enjoyed by the encoding of the π_a -calculus in cKLAIM (where, however, we have $\llbracket N \parallel M \rrbracket \equiv \llbracket N \rrbracket \parallel \llbracket M \rrbracket$) and by the encoding of KLAIM in μKLAIM . In this case we have a centralised entity (the locality env) that coordinates the translation of

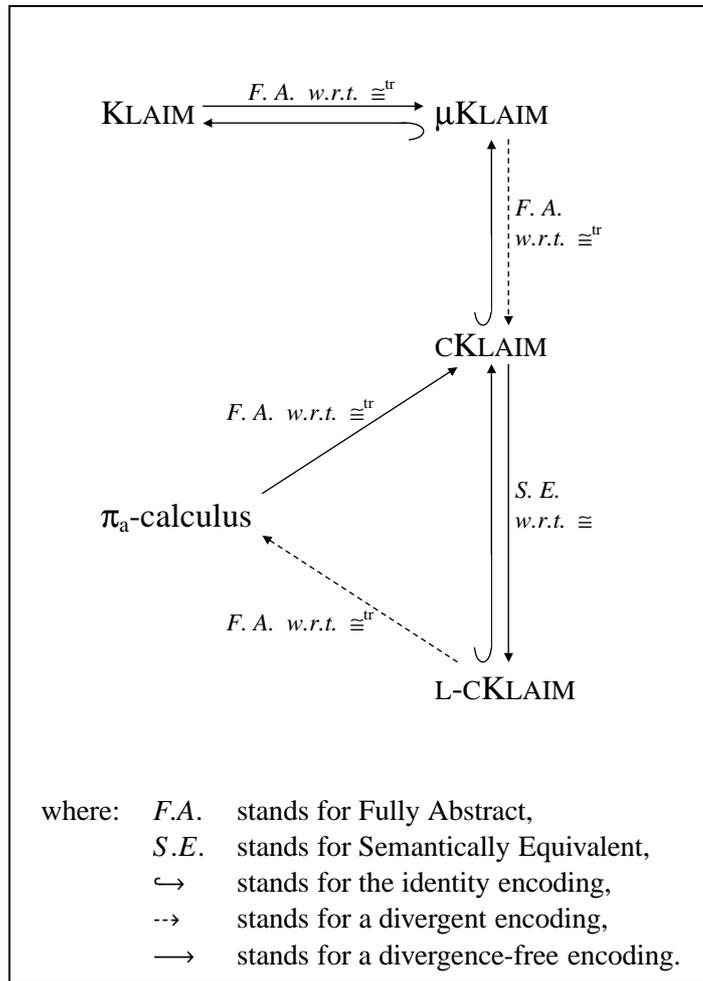


Table 6.8: Overview of our Results

names. According to Nestmann [118], the presence of such centralised authorities not necessarily implies that the encoding developed is weak: in practice, indeed, the resolution of names in Internet (through the so called DNS) requires some form of centralised knowledge to turn logical names in IP addresses. Hence, we believe that in this framework, property 1. is not strictly necessary.

Concluding assessment. We can now define a hierarchy of the properties enjoyed by our encodings. This can be done by relying on the properties pointed out in [122] and by the facts that \cong^{tr} is coarser than \cong and that semantical equivalence implies fully abstraction (w.r.t. the same equivalence). If we let ‘ \gg ’ mean ‘better than’, we have that

$$\xrightarrow{S.E. \text{ w.r.t. } \cong} \gg \xrightarrow{F.A. \text{ w.r.t. } \cong^{\text{tr}}} \gg \xrightarrow{F.A. \text{ w.r.t. } \cong^{\text{tr}}}$$

Thus, the encoding of cKLAIM in lcKLAIM is the best we can imagine: it does not introduce divergence and it turns a source net in a barbed congruent target net. The fact that barbed congruence is a fine-grained equivalence makes our result even stronger. Hence, the two calculi have exactly the same expressive power; the use of remote communications is only to ease programming.

The encoding of KLAIM in μKLAIM and of π_a -calculus in cKLAIM are still good results. Hence, the source and the target languages have comparable expressive power. Moreover, the complexity of the code generated by the encoding procedure is quite limited, especially in the second encoding.

The weaker results are the encodability of μKLAIM in cKLAIM and of lcKLAIM in π_a -calculus; this is not surprising since, as we already said, this is related to the expressive form of name matching we used. A part from divergence, we also have that the encoding of polyadic communication of μKLAIM into monadic one of cKLAIM is *not* simple and efficient. Table 6.2 substantiates this claim: a lot of monadic exchanges are necessary to implement each polyadic communication. While this could be acceptable from a theoretical point of view, it is hardly usable in practice. To conclude, we believe that, in a LINDA-like framework, these two forms of communication are not freely interchangeable. Hence, μKLAIM is rather more expressive than cKLAIM .

Clearly, the results in this paper do not prove that μKLAIM and lcKLAIM are more powerful than cKLAIM and π_a -calculus, respectively. To this aim, we should present some impossibility results similar to that in [122]. We are now working on proving an impossibility result for an encoding of lcKLAIM in the π_a -calculus; we believe that, due to the check of existence of the target of a communication (that is performed in lcKLAIM and not in π_a -calculus), this result should hold. About the encodability of a polyadic communication through monadic communications (μKLAIM into cKLAIM), we think that a divergence-free encoding should not exist. Indeed, the fields of a polyadic datum can only be accessed one by one; in this way, we are forced to split the atomic activity of function *match* into a field-by-field compliance checking. Such checking must be suspended whenever the datum accessed does not match with the template used to retrieve it, and the inputting process must be rolled back, to let it try to access another datum. Then, the possibility of repeatedly accessing the same (non-matching) datum clearly leads to divergence. Again, this is not a formal argument but we find it quite convincing.

Finally, as we have already said, \cong^{tr} is enough to state expressiveness results. On the other hand, an encoding can be also considered as a formal description of how a language primitive can be implemented by means of other primitives; then, the complexity of the encoding can be used as a measure of the power of the primitive. We plan to investigate this aspect in a future work. However, we can already say that \cong^{tr} will be too weak in such setting: preorders based on some notion of cost (maybe relying on whether an action is remote or not) should be used instead.

Related work. The works on encodings of process calculi that are strictly related to our approach have been discussed throughout the chapter. We want to conclude by examining the impact on expressiveness of alternative operators that have considered when designing KLAIM.

In [29] three different semantics for the output operation are studied in the setting of a simple Linda-based process calculus: *instantaneous output* (an output prefix immediately unleashes the corresponding tuple in the TS), *ordered output* (a reduction is needed to turn an output prefix into the corresponding tuple in the TS) and *unordered output* (two reductions are needed to turn an output into an available tuple, i.e. one to send the tuple to the TS and another one to make the tuple available in the TS). According to this terminology, the semantics of KLAIM output operation is ordered.

In [29] it is proved that the instantaneous semantics yields the most expressive setting. We believe that the instantaneous semantics would simplify the theory developed in this paper. For example, the proofs for the encoding of the π_a -calculus into cKLAIM would be simpler: the encoding would be prompt and no normalization would be needed. However, instantaneous tuple emission is unrealistic, especially in a global computing scenario where remote operations are enabled. On the other hand, unordered outputs are very close to the practice of global computers (consider, e.g., sending e-mail messages). We believe that the theory presented in this paper can be tailored to deal with such semantics. However, in [30] it is proved that the simple Linda-based process calculus considered in [29] is Turing powerful under the instantaneous and ordered semantics but not with the unordered semantics. The output operation of KLAIM represents a compromise between expressiveness and implementability.

The calculi considered in this paper do not have a general non-deterministic choice operator. This composition mechanism of ‘classical’ process calculi (e.g. CCS, CSP, ACP, π -calculus) is useful for specifying systems but is hardly implementable (this is especially true in distributed environments), thus it is missing in many languages designed while taking implementation issues into account. For example, the π_a -calculus is often presented without nondeterministic choice. From a theoretical point of view, one may wonder if the addition of (restricted forms of) the non-deterministic choice operator to the considered calculi would change their expressive power.

Since the same issue has been deeply studied in the setting of the π_a -calculus, let us first briefly summarize some relevant results. [119] presents two encodings of the π_a -calculus with input-guarded choice (where each summand begins with an input action) into ‘pure’ π_a -calculus, and [118] defines an encoding of a variant of the π -calculus with separate choice (where all summands begin with the same kind of action) in the π_a -calculus. [122] shows that no ‘reasonable’ encoding of the π -calculus with mixed choice (where each summand begins with an action) in the π_a -calculus can be ever given. Since synchronous communication can be encoded through asynchronous communication, [93, 19, 31], it follows that, in the setting of the π_a -calculus, the introduction of (some forms of) non-deterministic

choice changes the expressive power of the language. Similar conclusions can be drawn for the KLAIM-based languages we considered. By following [119, 118], we could implement restricted non-deterministic choice. However, we have to say that a restricted form of choice is provided implicitly by KLAIM and all its variants through actions **read/in** actions: their semantics is determined by the availability of tuples matching a given template, and in case of multiple matching the choice is internally determined.

Chapter 7

Conclusion and Future Work

This thesis collects some foundational work on calculi based on the paradigm put forward by the language KLAIM. To sum up, the main contributions contained in this thesis are:

- the definition of a family of foundational calculi based on KLAIM,
- the presentation of type systems for security driven by global computing requirements,
- the development of non-trivial, but still tractable, notions of behavioural equivalences,
- the assessment of the calculi presented from the point of view of their expressiveness.

The strength and the usefulness of these efforts should be witnessed by the examples provided throughout the thesis. In particular, we took into account several scenarios, ranging from e-commerce to on-line banking, from multiuser systems to distributed protocols. Moreover, the encodings used to carry on the analysis of the expressive power of the calculi should also have improved the understanding of the KLAIM and the appreciation of its specific design choices, that make it significantly different from the standard process calculi and from the other calculi for GC.

Further developments. While writing this thesis, we extended the work on KLAIM-derived calculi with several issues like *dynamic inter-node connections* [60], *failures* [63] and more flexible forms of pattern-matching [64]. We did not include here such results to save space. However, it is worth saying that the work we have just presented was the basis for all these enhancements. This fact furtherly substantiates the claim that this thesis acts as a crucial foundational work within the KLAIM project.

Future work. In addition to the work covered by this thesis, we studied other possible approaches to security in calculi with mobility, that could be integrated with the work presented here. We now conclude by discussing the most interesting issues.

In Chapter 3, we expressed node policies in a very simple way, essentially as sets of process actions. More sophisticated possibilities are available. For example, we could exploit the theory developed in [23], where we study the impact of role-based access control (RBAC, [132]) mechanisms within a distributed π -calculus. The policies we used in this paper can be defined in a more flexible way by using the RBAC approach. Moreover, in [78] we work on a very simple distributed calculus with code mobility where several fine-grained policies are expressed by using, e.g., finite automata. By properly modifying the types presented in Chapter 3, we believe that the theory developed in *loc.cit.* can be adapted to the KLAIM setting.

A concrete issue that we totally ignored in this thesis is the use of *cryptograpy*. As we have already remarked several times, cryptography is a fundamental lower-level tool to ensure that GCs work properly. Thus, in [17, 18], we worked on the Spi-calculus [1], a cryptographic version of the π -calculus, to develop a sound and complete axiomatisation for finite processes under a bisimulation-based semantics; then, we used some equational laws to show the correctness of the protocol KERBEROS. Clearly, cryptography is an orthogonal issue with respect to the topics covered by this thesis. A challenging direction for future research is to study the integration of cryptographic primitives in a calculus with mobile agents, like KLAIM.

Appendix A

Symbols and Notations

In Chapter 2:

- \mathcal{L} : locality names (in KLAIM)
- \mathcal{V} : locality variables (in KLAIM)
- \mathcal{N} : names (in μKLAIM , cKLAIM and LCKLAIM)
- ρ : allocation environments (finite mappings of names for variables)
- $fn(\cdot), bn(\cdot)$: free and bound names
- $fv(\cdot), bv(\cdot)$: free and bound variables (in KLAIM)
- $\mathcal{E}[\![\cdot]\!]_{\rho}$: the evaluation function of a tuple/template w.r.t. ρ
- $match$: the pattern-matching function (between a template and a tuple)
- σ :
 - . substitutions of names and processes for variables in KLAIM
 - . substitutions of names for names in μKLAIM , cKLAIM and LCKLAIM
- ϵ : the empty substitution
- \circ : the composition of substitutions
- \equiv : the structural congruence relation
- \mapsto : the reduction relation

In Chapter 3:

- \mathcal{C} : the set of process capabilities
- π : sets of capabilities
- Π : the powerset of \mathcal{C} (i.e., the set of all the π s)
- \sqsubseteq_{Π} : an ordering on Π
- \uplus : the pointwise union of functions
- Γ : typing environments
- upd : the extension of a type environment with the typing annotations for the names bound in a template
- \nearrow : the run-time error predicate

In Sections 3.2 and 3.3:

- δ : types and type environments
- \leq : the subtyping relation
- $match_\delta$: the typed pattern-matching function

In Section 3.4:

- p : a finite set of template patterns
- \mathcal{P} : the set of all template patterns
- $\mathcal{T}(\cdot)$: the set of all templates complying with a set of patterns
- Δ : types
- \perp : the empty type
- \leq : the subtyping relation
- $match_{\Delta(t)}$: the typed pattern-matching function

In Chapter 4:

- r : regions (i.e., finite subsets of \mathcal{L})
- \top : all the net (i.e., the largest region)
- \mathcal{R} : the set of all regions
- $>$: the type annotation procedure
- \uplus : the union of functions with disjoint domains
- $+$: the extension of information in a typing environment
- $reg(\cdot)$: a function returning the intersection of regions in \cdot
- $- \nearrow^S$: turning each region r in \top within a typing environment $-$,
if $r \cap S \neq \emptyset$
- \gg : tagged typing annotation
- \mapsto : tagged reduction relation

In Chapter 5:

- \downarrow, \Downarrow : strong and weak basic observable (or barb)
- $C[\cdot]$: net context
- \cong : reduction barbed congruence
- \approx : may testing
- \approx : (labelled) bisimulation
- $\vec{\rightarrow}, \vec{\Rightarrow}$: strong and weak labelled transitions
- τ : silent action
- I : inert node components
- α : labels for the bisimulation
- \times : trace equivalence
- μ : labels for the trace equivalence
- ϱ : traces (i.e., sequences of visible actions)
- ϵ : empty trace
- \cdot : concatenation of traces
- \leq : ordering relation on traces
- $\bar{\cdot}$: complementation on traces
- $\Pi(\varrho)$: canonical observer for trace ϱ

In Chapter 6:

- \succsim : expansion preorder
- \equiv^{tr} : translated reduction barbed congruence
- \succsim^{tr} : translated expansion preorder
- $\langle\langle \cdot \rangle\rangle$: encoding KLAIM in μKLAIM
- $\langle\langle\langle \cdot \rangle\rangle\rangle$: normalisation of $\langle\langle \cdot \rangle\rangle$
- $\langle \cdot \rangle$: encoding μKLAIM in cKLAIM
- $\{\{ \cdot \}\}$: encoding cKLAIM in LCKLAIM
- $\{\{\{ \cdot \}\}\}$: normalisation of $\{\{ \cdot \}\}$
- $\llbracket \cdot \rrbracket$: encoding π_a -calculus in cKLAIM
- $\llbracket\llbracket \cdot \rrbracket\rrbracket$: normalisation of $\llbracket \cdot \rrbracket$
- (\cdot) : encoding LCKLAIM in π_a -calculus

Bibliography

- [1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999. An extended abstract appeared in the *Proc. of the 4th ACM Conf. on Computer and Communications Security*, 1997.
- [2] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of LNCS, pages 111–130. Springer, 1997.
- [3] G. Agha. *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [4] R. Amadio. On modelling mobility. *Theoretical Computer Science*, 240(1):147–176, 2000.
- [5] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π -calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proc. of CONCUR '96*, LNCS 1119: 147–162.
- [6] K. Arnold, E. Freeman, and S. Hupfer. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.
- [7] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
- [8] H. P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1984 (revised edition).
- [9] L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming and their Implementations*. PhD thesis, Dip. di Matematica, Univ. di Siena, 2003.
- [10] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM project: Theory and practice. In C. Priami, editor, *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, number 2874 in LNCS, pages 88–150. Springer, 2003.
- [11] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In P. Ciancarini and R. Tolksdorf, editors, *Proc. of the 7th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 110–115. IEEE Computer Society, 1998.
- [12] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Mobile Code. *Software — Practice and Experience*, 32:1365–1394, 2002.

- [13] M. Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Comput. Sci.*, 195(2):205–226, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 163–178.
- [14] M. Boreale and R. De Nicola. Testing equivalences for mobile processes. *Information and Computation*, 120:279–303, 1995. An extended abstract appeared in *Proc. of CONCUR '92*, LNCS 630.
- [15] M. Boreale, R. De Nicola, and R. Pugliese. Basic observables for processes. *Information and Computation*, 149(1):77–98, 1999.
- [16] M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172:139–164, 2002.
- [17] M. Boreale and D. Gorla. On compositional reasoning in the spi-calculus. In M. Nielsen and U. Engberg, editors, *Proceedings of FoSSaCS '02*, volume 2303 of LNCS, pages 67–81. Springer-Verlag, 2002.
- [18] M. Boreale and D. Gorla. Process calculi and the verification of security properties. *Journal of Telecommunication and Information Technology— Special Issue on Cryptographic Protocol Verification*, (4):28–40, 2002.
- [19] G. Boudol. Asynchrony and the π -calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.
- [20] G. Boudol. Typing the use of resources in a concurrent calculus. In R. K. Shyamasundar and K. Ueda, editors, *Proceedings of ASIAN '97*, volume 1345 of LNCS, pages 239–253. Springer, 1997.
- [21] G. Boudol, I. Castellani, F. Germain, and M. Lacoste. Models of distribution and mobility: state of the art. MIKADO Global Computing Project, IST-2001-32222. Deliverable D1.1.1, reference RR/WP1/1. 2002.
- [22] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. A theory of processes with localities. *Formal Aspects of Computing*, 6:165–200, 1994.
- [23] C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *Proceedings of 17th CSFW*. IEEE Computer Society, 2004. Full version as Tech. Rep. 08/2004, Dip. di Informatica, Univ. di Roma ‘La Sapienza’.
- [24] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proceedings of POPL '01*. ACM, 2001.
- [25] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proceedings of TACS '01*, number 2215 in LNCS, pages 38–63. Springer, 2001.
- [26] M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *Proceedings of CONCUR '01*, number 2154 in LNCS, pages 102–120. Springer, 2001.
- [27] M. Bugliesi, D. Colazzo, and S. Crafa. Types for discretionary access control. In *Proceedings of CONCUR'04*, number 3170 in LNCS, pages 225–239. Springer, 2004.
- [28] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication interference in mobile boxed ambients. In M. Agrawal and A. Seth, editors, *Proceedings of FSTTCS'02*, volume 2556 of LNCS, pages 71–84. Springer, 2002.

- [29] N. Busi, R. Gorrieri, and G. Zavattaro. Comparing three semantics for linda-like languages. *Theoretical Computer Science*, 240(1):49–90, 2000.
- [30] N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of linda coordination primitives. *Information and Computation*, 156(1-2):90–121, 2000.
- [31] D. Cacciagrano and F. Corradini. On synchronous and asynchronous communication paradigms. In *Proc. of ICTCS'01*, number 2202 in LNCS, pages 256–268. Springer, 2001.
- [32] L. Caires and L. Cardelli. A Spatial Logic for Concurrency. *Information and Computation*, 186(2):194–235, 2003. Earlier, this work appeared split in two parts, respectively in *Proc. of TACS'01*, LNCS 2215, and *Proc. of CONCUR'02*, LNCS 2421.
- [33] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.
- [34] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 51–94. Springer, 1999.
- [35] L. Cardelli, G. Ghelli, , and A. D. Gordon. Secrecy and group creation. In *Proc. of CONCUR'00*, number 1877, pages 365–379. Springer, 2000.
- [36] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In J. Wiederman, P. van Emde Boas, and M. Nielsen, editors, *Proc. of ICALP'99*, volume 1644 of LNCS, pages 230–239. Springer, 1999.
- [37] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Proc. of IFIP-TCS'00*, volume 1872 of LNCS, pages 333–347. Springer, 2000.
- [38] L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the ambient calculus. *Information and Computation*, 177(2):160–194, 2002.
- [39] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proc. of POPL'99*, pages 79–92. ACM, 1999.
- [40] L. Cardelli and A. D. Gordon. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *Proc. of POPL'00*. ACM, 2000.
- [41] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Comput. Sci.*, 240(1):177–213, 2000. An extended abstract appeared in *Proc. of FoSSaCS '98*, number 1378 of LNCS, pages 140-155.
- [42] L. Cardelli and A. D. Gordon. Logical Properties of Name Restriction. In S. Abramsky, editor, *Proc. of TLCA'01*, number 2044 in LNCS. Springer, 2001.
- [43] G. Castagna, G. Ghelli, and F. Z. Nardelli. Typing mobility in the seal calculus. In *Proceedings of CONCUR '01*, number 2154 in LNCS, pages 82–101. Springer, 2001.
- [44] G. Castagna and F. Z. Nardelli. The Seal Calculus Revisited: contextual equivalence and bisimilarity. In M. Agrawal and A. Seth, editors, *Proceedings of FSTTCS'02*, volume 2556 of LNCS, pages 85–96. Springer, 2002.

- [45] G. Castagna and J. Vitek. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in LNCS, pages 47–77. Springer, 1999.
- [46] I. Castellani. Process algebras with localities. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 945–1045. Elsevier Science, 2001.
- [47] I. Castellani and M. Hennessy. Distributed bisimulations. *Journal of the ACM*, 36:887–911, 1989.
- [48] I. Castellani and M. Hennessy. Testing theories for asynchronous languages. In V. Arvind and R. Ramanujam, editors, *Proceedings of FSTTCS '98*, volume 1530 of LNCS, pages 90–101, 1998.
- [49] S. Castellani, P. Ciancarini, and D. Rossi. The ShaPE of ShaDE: a coordination system. Technical Report UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. di Bologna, Italy, 1996.
- [50] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–366, 1998.
- [51] R. Cleaveland. Tableau-based model checking in the propositional μ -calculus. *Acta Informatica*, 27(8):725–747, 1990.
- [52] Concurrency and Mobility Group at Dipartimento di Sistemi e Informatica, Università di Firenze. KLAIM web page. <http://music.dsi.unifi.it>.
- [53] M. Coppo, M. Dezani, E. Giovannetti, and R. Pugliese. Dynamic and Local Typing for Mobile Ambients. In *Proc. of IFIP-TCS'04*. Kluwer, 2004.
- [54] F. Corradini and R. De Nicola. Locality based semantics for process algebra. *Acta Informatica*, 34:291–324, 1997.
- [55] S. Dal Zilio and A. D. Gordon. Region analysis and a pi-calculus with groups. In *Proc. of MFCS'00*, number 1893, pages 409–424. Springer, 2000.
- [56] N. Davies, S. Wade, A. Friday, and G. Blair. L²imbo: a tuple space based platform for adaptive mobile applications. In *Int. Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP'97)*, 1997.
- [57] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
- [58] R. De Nicola, G. Ferrari, and R. Pugliese. Programming Access Control: The Klaim Experience. In C. Palamidessi, editor, *Proceedings of CONCUR'00*, volume 1877 of LNCS, pages 48–65. Springer, 2000.
- [59] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
- [60] R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. Technical report. 07/2004, Dip. di Informatica, Univ. di Roma 'La Sapienza'. An extended abstract will appear in *Proc. of ICALP'05*.

- [61] R. De Nicola, D. Gorla, and R. Pugliese. Confining data and processes in global computing applications. *Science of Computer Programming*. To appear.
- [62] R. De Nicola, D. Gorla, and R. Pugliese. On the expressive power of KLAIM-based calculi. *Proc. of EXPRESS'04, ENTCS*, 128(2):117–130, 2004. Full version as Tech. Rep. 09/2004, Dip. di Informatica, Univ. di Roma ‘La Sapienza’; to appear in *Theoretical Computer Science*.
- [63] R. De Nicola, D. Gorla, and R. Pugliese. Global computing in a dynamic network of tuple spaces. In J. Jacquet and G. Picco, editors, *Proc. of 7th International Conference on Coordination Models and Languages (COORDINATION 2005)*, volume 3454 of *LNCS*, pages 157–172. Springer, 2005.
- [64] R. De Nicola, D. Gorla, and R. Pugliese. Pattern matching over a dynamic network of tuple spaces. In M. Steffen and G. Zavattaro, editors, *Proc. of 7th IFIP International Conference on Formal Methods for Object Oriented-based Distributed Systems (FMOODS 2005)*, volume 3535 of *LNCS*, pages 1–14. Springer, 2005.
- [65] R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
- [66] R. De Nicola and M. Loreti. A Modal Logic for Mobile Agents. *ACM Transactions on Computational Logic*, 2004. An extended abstract appeared in *Proc. of AMAST'00*, *LNCS* 1816.
- [67] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Proc. of ASIAN'00*, volume 1961 of *LNCS*, pages 199–214. Springer, 2000.
- [68] D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proceedings of ISADS '01*, pages 278–286. IEEE Computer Society, 2001.
- [69] M. Dezani-Ciancaglini and I. Salvo. Security types for mobile safe ambients. In *Proceedings of ASIAN'00*, volume 1961 of *LNCS*, pages 215–236. Springer, 2000.
- [70] G. M. F. B. Schneider and R. Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back. Conference on the Occasion of Dagstuhl's 10th Anniversary*, volume 2000 of *LNCS*, page 86. Springer, 1989.
- [71] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
- [72] C. Fournet and C. Laneve. Bisimulations for the join-calculus. *Theoretical Computer Science*, 266(1-2):569–603, 2001.
- [73] A. Fuggetta, G. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.
- [74] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
- [75] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Proceedings of PARLE '89*, volume 366 of *LNCS*, pages 20–27. Springer-Verlang, 1989.
- [76] General Magic, Inc. Odyssey, 1998. <http://www.genmagic.com/-technology/odyssey.html>.

- [77] J. Godskesen, T. Hildebrandt, and V. Sassone. A calculus of mobile resources. In L. Brim, P. Jancar, M. Kretínský, and A. Kucera, editors, *Proceedings of CONCUR'02*, volume 2421 of *LNCS*, pages 272–287. Springer, 2002.
- [78] D. Gorla, M. Hennessy, and V. Sassone. Security policies as membranes in systems for global computing. In J. Rathke, editor, *Proc. of 3rd EATCS Workshop on Foundations of Global Ubiquitous Computing (FGUC'04)*, ENTCS. Elsevier, 2004. Full version as Research Rep. 02/2004, Dep. Informatics, Univ. of Sussex; to appear in *Logical Methods in Computer Science*.
- [79] D. Gorla and R. Pugliese. Enforcing security policies via types. In D. Hutter, G. Mueller, W. Stephan, and M. Ullman, editors, *Proc. of 1st Intern. Conf. on Security in Pervasive Computing (SPC'03)*, volume 2802 of *LNCS*, pages 88–103. Springer-Verlag, 2003. Full version as Tech. Rep. 05/2004, Dip. di Informatica, Univ. di Roma ‘La Sapienza’.
- [80] D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proceedings of ICALP'03*, volume 2719 of *LNCS*, pages 119–132. Springer-Verlag, 2003. Full version as Tech. Rep. 06/2004, Dip. di Informatica, Univ. di Roma ‘La Sapienza’.
- [81] D. Gorla and R. Pugliese. Controlling data movement in global computing applications. In *Proc. of the 19th ACM-SIGAPP Symposium on Applied Computing (SAC'04)*, pages 1462–1467. ACM, 2004.
- [82] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. *PhD Thesis*. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, 1997.
- [83] R. R. Hansen, J. G. Jensen, F. Nielson, and H. R. Nielson. Abstract interpretation of mobile ambients. In *Proc. of SAS'99*, number 1694 in *LNCS*, pages 134–148. Springer, 1999.
- [84] M. Hennessy. The security pi-calculus and non-interference. In *Proc. of MFPS XIX*, ENTCS, 2003. Full version to appear in *Journal of Logic and Algebraic Programming*.
- [85] M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *Proceedings of FoSSaCS '03*, volume 2620 of *LNCS*, pages 282–299. Springer, 2003. Full version as COGS Computer Science Technical Report, 2002:01.
- [86] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, 1985.
- [87] M. Hennessy, J. Rathke, and N. Yoshida. SafeDpi: a language for controlling mobile code. In *Proceedings of FoSSaCS'04*, volume 2987 of *LNCS*, pages 241–256. Springer, 2004.
- [88] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 111–130. Springer, 1997.

- [89] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In U. Montanari, J. Rolim, and E. Welzl, editors, *Proceedings of ICALP'00*, volume 1853 of *LNCS*, pages 415–427. Springer, 2000. Full version to appear in *ACM TOPLAS*.
- [90] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.
- [91] C. Hoare. Communicating sequential processes. In R. McKeag and A. Macnaghten, editors, *On the construction of programs – an advanced course*, pages 229–254. Cambridge university press, 1980.
- [92] K. Honda. Types for dyadic interaction. In E. Best, editor, *Proceedings of CONCUR '93*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.
- [93] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.
- [94] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In G. Smolka, editor, *Proceedings of ESOP '00*, volume 1782 of *LNCS*, pages 180–199. Springer, 2000.
- [95] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Proceedings of ESOP '98*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
- [96] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995. An extract appeared in *Proc. of FSTTCS '93*, LNCS 761.
- [97] D. Johansen, F. Schneider, and R. Renesse. What TACOMA Taught Us. In D. Milojicic, F. Douglass, and R. Wheeler, editors, *Mobility, Mobile Agents and Process Migration - An edited Collection*. Addison-Wesley, 1998.
- [98] D. Kirli. Confined mobile functions. In *Proceedings of 14th CSFW*. IEEE Computer Society, 2001.
- [99] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transaction on Programming Languages and Systems*, 20(2):436–482, 1998. An extended abstract previously appeared in the *Proc. of LICS '97*, pages 128–139.
- [100] N. Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Proceedings of IFIP TCS 2000*, volume 1872 of *LNCS*, pages 365–389. Springer, 2000.
- [101] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings of POPL '96*, pages 358–371. ACM, 1996.
- [102] N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In C. Palamidessi, editor, *Proceedings of CONCUR '00*, volume 1877 of *LNCS*, pages 489–503. Springer, 2000.
- [103] D. Lange and M. Oshima. *Programming and Deploying Java mobile Agents with Aglets*. Addison-Wesley, 1998.

- [104] F. Levi and S. Maffei. An abstract interpretation framework for analysing mobile ambients. In *Proc. of SAS'01*, number 2126 in LNCS, pages 395–411. Springer, 2001.
- [105] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL '00*, pages 352–364. ACM, 2000.
- [106] M. Loreti. *Languages and Logics for Network Aware Programming*. PhD thesis, Università di Siena, 2002. Available at <http://music.dsi.unifi.it>.
- [107] M. Merro and M. Hennessy. Bisimulation congruences in Safe Ambients. In *Proceedings of POPL '02*. ACM, 2002.
- [108] M. Merro and F. Z. Nardelli. Bisimulation proof methods for mobile ambients. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proceedings of ICALP'03*, volume 2719 of LNCS. Springer, 2003.
- [109] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
- [110] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [111] R. Milner. Sorts in the π -calculus (extended abstract). In E. Best and G. Rozenberg, editors, *Proc. of the 3rd Workshop on Concurrency and Compositionality*, volume 191 of *GMD-Studien*. GMD Bonn, St. Augustin, 1991.
- [112] R. Milner. The polyadic π -calculus: A tutorial. In F. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F: Computer and System Sciences*. NATO Advanced Study Institute, 1993.
- [113] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.
- [114] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.
- [115] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of LNCS, pages 685–695. Springer, 1992.
- [116] U. Montanari and M. Pistore. Finite state verification for the asynchronous π -calculus. In R. Cleaveland, editor, *Proc. of TACAS'99*, volume 1579 of LNCS, pages 255–269. Springer, 1999.
- [117] G. Necula. Proof-carrying code. In *Proc. of POPL '97*, pages 106–119, 1997.
- [118] U. Nestmann. What is a ‘good’ encoding of guarded choice? *Information and Computation*, 156:287–319, 2000. An extended abstract appeared in the *Proceedings of EXPRESS '97*, volume 7 of *ENTCS*.
- [119] U. Nestmann and B. C. Pierce. Decoding choice encodings. *Information and Computation*, 163:1–59, 2000. An extended abstract appeared in the *Proceedings of CONCUR '96*, LNCS 1119, pages 179–194.
- [120] F. Nielson, H. R. Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In J. C. Baeten and S. Mauw, editors, *Proceedings of CONCUR '99*, volume 1664 of LNCS, pages 463–477. Springer, 1999.
- [121] A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):251–269, 1999.

- [122] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π -calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. An extended abstract appeared in *Proc. of POPL'97*, ACM Press.
- [123] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *5th GI Conference*, volume 104 of *LNCS*. Springer, 1981.
- [124] J. Parrow. An introduction to the pi-calculus. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.
- [125] G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM, 1999.
- [126] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extract appeared in *Proc. of LICS '93*: 376–385.
- [127] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
- [128] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL '98*. ACM, 1998.
- [129] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL '99*, pages 93–104. ACM, 1999. Full version to appear in *Journal of Automated Reasoning*, 2003.
- [130] J. Riely and M. Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 266:693–735, 2001.
- [131] A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.
- [132] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
- [133] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, 1993. CST-99-93 (also published as ECS-LFCS-93-266).
- [134] D. Sangiorgi. Bisimulation in higher-order process calculi. *Information and Computation*, 131:141–178, 1996. An early version appeared in *Proc. of PROCOMET'94*, pages 207–224.
- [135] D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999. An abstract appeared in the *Proc. of ICALP '97*, LNCS 1256, pages 303–313.
- [136] D. Sangiorgi. Reasoning about concurrent systems using types. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *LNCS*, pages 31–40. Springer, 1999.
- [137] D. Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253(2):311–350, 2001.

- [138] D. Sangiorgi and D. Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- [139] P. Sewell. Global/local subtyping and capability inference for a distributed pi-calculus. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *LNCS*, pages 695–706. Springer, 1998.
- [140] P. Sewell, P. Wojciechowski, and B. Pierce. Location independence for mobile agents. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Proceedings of ICCL '98*, volume 1686 of *LNCS*. Springer, 1999.
- [141] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.
- [142] P. Thati, R. Ziaei, and G. Agha. A Theory of May Testing for Actors. In *Proc. of FMOODS'02*, pages 147–162. Kluwer, 2002.
- [143] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 278–298. Springer, 1996.
- [144] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, 1996. CST-126-96 (also published as ECS-LFCS-96-345).
- [145] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructures for mobile computation. In *Proceedings of POPL '01*, pages 116–127. ACM, 2001.
- [146] R. van Glabbeek. The linear time - branching time spectrum. In J. Baeten and J. Klop, editors, *Proceedings of CONCUR'90*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.
- [147] R. van Glabbeek. The linear time - branching time spectrum II; the semantics of sequential systems with silent moves. In E. Best, editor, *Proceedings of CONCUR'93*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.
- [148] J. E. White. Telescript Technology: The Foundation for the Electronic Marketplace. White paper, General Magic, Inc., Mountain View, CA, 1994.
- [149] J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.
- [150] G. Winskel. A note on model checking the modal ν -calculus. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Proc. of the 16th ICALP*, volume 372 of *LNCS*, pages 761–772. Springer, 1989.
- [151] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. Tspaces. *IBM Systems Journal*, 37(3):454–474, 1998.
- [152] N. Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *LNCS*, pages 371–386. Springer, 1996.
- [153] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In J. C. Baeten and S. Mauw, editors, *Proceedings of CONCUR '99*, volume 1664 of *LNCS*, pages 557–572. Springer, 1999.
- [154] N. Yoshida and M. Hennessy. Assigning types to processes. *Information and Computation*, 174(2):143–179, 2002. An extended abstract appeared in *Proc. of LICS'00*, pagg. 334–348.