Dynamic Web Data: a Process Algebraic Approach

Sergio Maffeis
Department of Computing, Imperial College London
maffeis@doc.ic.ac.uk

Ph.D. Thesis, 2005.
Thesis supervisor: Philippa Gardner
Examiners: Matthew Hennessy and Val Tannen

Abstract

Peer to peer systems, exchanging dynamic documents through Web services, are a simple and effective platform for data integration on the internet. Dynamic documents can contain both data and references to external sources in the form of links, calls to web services, or coordination scripts. XML standards, and industrial platforms for web services, provide the technological basis for building such systems. We argue that process algebras are a promising tool for studying and understanding their formal properties.

In this thesis, we define the X$d\pi$-calculus with the aim of reasoning about dynamic Web data. X$d\pi$ terms represent networks of peers, each consisting of an XML data repository and a working space where processes are allowed to run. Processes, inspired by the π-calculus, can communicate with each other, query and update the local repository, or migrate to other peers to continue execution. Data can contain scripted processes, which can be executed by other processes. For example, X$d\pi$ processes can be used to embed service calls in documents and to model Web services.

We investigate behavioural equivalences for X$d\pi$, comparing several observable properties, such as the shape of data trees and the communication actions attempted by processes. To simplify reasoning on equivalences, we introduce Core X$d\pi$, a calculus which is semantically equivalent to X$d\pi$, but where processes are located explicitly and are separated from the data repository.

To help proving equivalences, which require a costly property of closure under contexts, we define a coinductive relation (called *domain bisimilarity*) which does not quantify over contexts and which entails process equivalence. Its definition is non-standard, because scripts are part of the values, and process equivalences are sensitive to the set of locations constituting the network. We apply bisimilarity to study some communication patterns used by servers in distributed query systems, and we propose a new pattern involving mobile code.

Declaration about conjoint work

The composition of the thesis is entirely my own. The X$d\pi$ model and the behavioural equivalences as presented in this thesis, are a revised and extended version of the ones presented in earlier work done in collaboration with my supervisor Philippa Gardner. The technical development of the behavioural equivalences is mostly my own.

The design of Core X$d\pi$ is inspired by previous work on polyadic synchronization done in collaboration with Marco Carbone.

Contents

Main Notations **6**

1 Introduction **8**
1.1 Dynamic Web data . . . 8
1.2 A process algebraic approach . . . 10
1.3 Related work . . . 13
1.3.1 Distributed query processing . . . 13
1.3.2 Web scripting languages . . . 15
1.3.3 Process algebras for service orchestration and XML . . . 15

2 The X$d\pi$-calculus **18**
2.1 X$d\pi$ informally . . . 18
2.1.1 The data model . . . 18
2.1.2 Processes and networks . . . 20
2.2 Syntax and semantics . . . 22
2.2.1 Trees, data and queries . . . 22
2.2.2 Processes and networks . . . 24
2.2.3 Semantics . . . 27
2.3 A sample query and update language . . . 28

3 Examples **31**
3.1 Derived constructs . . . 31
3.1.1 Control . . . 32
3.1.2 Data . . . 34
3.2 Web services . . . 35
3.3 Dynamic data . . . 37
3.3.1 Bibliographic database . . . 37
3.3.2 Active XML and beyond . . . 39
3.3.3 Stock quoting service . . . 40
3.3.4 Market indexes . . . 42

4 Behavioural Equivalences **44**
4.1 Equivalences and observables . . . 44
4.1.1 Induced reduction congruence . . . 44
4.1.2 Limit observables . . . 47
4.2 Network equivalences . . . 47

4.2.1 X$d\pi$ observables . . . 47
4.2.2 Comparing the equivalences . . . 51
4.3 Core X$d\pi$ . . . 56
4.3.1 Syntax and semantics . . . 56
4.3.2 From X$d\pi$ to Core X$d\pi$ . . . 59
4.3.3 Properties of the encoding . . . 62
4.4 Process equivalences . . . 68

5 Bisimilarity **71**
5.1 Labelled transition system . . . 71
5.2 Domain bisimilarity . . . 77
5.3 Results and proofs . . . 79
5.3.1 Basic properties . . . 79
5.3.2 Congruence . . . 83
5.3.3 Soundness . . . 96
5.3.4 Fixed-point characterization . . . 101

6 Distributed Query Patterns **103**
6.1 Chaining, recruiting and referral . . . 103
6.1.1 Implementing the patterns . . . 105
6.1.2 Relating the patterns to a specification . . . 106
6.2 Rendez-vous and shipping . . . 109
6.2.1 The rendez-vous query pattern . . . 110
6.2.2 Equivalence of the patterns . . . 111

7 Concluding Remarks **116**
7.1 Review . . . 116
7.1.1 Publication history . . . 119
7.2 Future work . . . 120

Bibliography **121**

A Tables **128**

List of Figures

1.1 Reference architecture 9

2.1 Reference architecture 19
2.2 Representing XML in X$d\pi$ 20
2.3 Syntax: trees and data 23
2.4 Syntax: X$d\pi$ processes 25
2.5 Function *cval* and predicate *distinct* 26
2.6 Syntax: X$d\pi$ networks 26
2.7 Semantics: structural congruence for X$d\pi$ 27
2.8 Semantics: reduction relation for X$d\pi$ 28
2.9 Syntax: Sam queries 29
2.10 Semantics: query evaluation for Sam 30

3.1 Notation and conventions for the derived constructs 32

4.1 Shape equivalence 49
4.2 Syntax: Core X$d\pi$ processes 57
4.3 Syntax: Core X$d\pi$ networks 58
4.4 Semantics: structural congruence for Core X$d\pi$ 58
4.5 Semantics: reduction relation for Core X$d\pi$ 59
4.6 Encodings from X$d\pi$ to Core X$d\pi$ 60
4.7 Notation for asynchronous processes 69

5.1 Syntax: configurations 72
5.2 Function *triggers* and predicate *unique*. 72
5.3 Functions *dom* and *scripts* for configurations 73
5.4 Extraction relation 74
5.5 Labels for the transition system 75
5.6 Labelled transition system 76
5.7 Merge operator for configurations 84

6.1 Chaining, recruiting and referral 104
6.2 Combining the query patterns 104
6.3 Rendez-vous 110
6.4 Bisimulation Diagrams 113

A.1 Function *dom* for X$d\pi$ networks 128
A.2 Function *dom* for Core X$d\pi$ 128

A.3 Full structural congruence for X$d\pi$ 129
A.4 Free variables and free names for X$d\pi$ 130
A.5 Full structural congruence for Core X$d\pi$ 131
A.6 Free variables and free names for Core X$d\pi$ 132

Main Notations

We report below some standard notions and syntactic conventions which will be used in the rest of the thesis.

Tuples. Let t be an arbitrary syntactic term. We denote by ε the empty tuple, and we abbreviate the tuple $t_1, \ldots, t_n$ (where $n \geq 0$) with $\widetilde{t}$.

Substitutions and binders. For each kind of term, we will define a function fv returning the set of its *free variables*. A term t is *open* if it contains free variables and it is *closed* otherwise. We use italic letters for closed terms (e.g. t) and italic bold letters for arbitrary terms (e.g. $\boldsymbol{t}$) when it is important to distinguish between them. A variable is *fresh* with respect to a term if it does not belong to its free variables.

A *substitution* is a finite map σ from variables to terms, which we represent in set-theoretic notation as $\sigma = \{{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}\}$, where $\mathit{domain}(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = t_i$. We abbreviate with the substitution $\{{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}\}$ with $\{{}^{\widetilde{t}}/_{\widetilde{x}}\}$. We define the *application* $t\sigma$ of a substitution σ to a term t as the simultaneous replacement in t of all the free occurrences of each x_i in $\mathit{domain}(\sigma)$ with the corresponding t_i. We say that σ is a *closing* substitution for t if $t\sigma$ is closed. A substitution σ is *well-sorted* if for each x_i in $\mathit{domain}(\sigma)$, $\sigma(x_i)$ and x_i belong to the same sort.

We enclose binders in small round brackets, and we let their scope extend to the right: for example, $(x)t$ is a term where the variable x is a binder with scope t. We identify terms with bound variables up-to α-conversion, and we assume that substitution always avoids capturing bound variables: for example, $(x)t\{{}^{x}/_{y}\} = (z)(t\{{}^{z}/_{x}\})\{{}^{x}/_{y}\}$.

Sets. Let $\mathcal{S} = \{t_1, \ldots, t_n\}$ be and arbitrary set. We denote by $\cup, \cap, \setminus, {}^{-1}$ the usual operations of set union, intersection, difference and complement, and we use the abbreviations $\mathcal{S} + t \stackrel{\text{def}}{=} \mathcal{S} \cup \{t\}$ for the union of a set with a singleton, and $\mathcal{S} - t \stackrel{\text{def}}{=} \mathcal{S} \setminus \{t\}$ for the difference. We denote the powerset of $\mathcal{S}$ by $\wp(\mathcal{S})$ and the cartesian product $\mathcal{S} \times \mathcal{S}^n$ by $\mathcal{S}^{n+1}$ (where $\mathcal{S}^1 = \mathcal{S}$).

Lists. Let L an arbitrary list. We denote by $\varnothing$ the empty list, by $t \shortmid L$ the list with head t and tail L, by $L \shortmid\shortmid L'$ the concatenation of L and L', and by $t_1 \shortmid \ldots \shortmid t_n \shortmid \varnothing$ the list with elements $t_1, \ldots, t_n$. We often omit a trailing $\varnothing$. We will interpret t as a term, except when by the context we can infer that it is the list $t \shortmid \varnothing$. We denote by $\mathcal{S}^{\prec\omega}$ the set of finite lists over $\mathcal{S}$.

Relations. A binary relation $\mathcal{R}$ over $\mathcal{S}, \mathcal{S}'$ is a subset of $\mathcal{S} \times \mathcal{S}'$. We denote the fact that t and t' are in $\mathcal{R}$ indifferently by $(t, t') \in \mathcal{R}$, or $\mathcal{R}(t, t')$, or $\mathcal{R}(t) = t'$, or $t\mathcal{R}t'$. If it is not the case that $t\mathcal{R}t'$ we write $t\not\mathcal{R}t'$, and if there is no t' such that $t\mathcal{R}t'$ we write $t\not\mathcal{R}$ (and similarly for $\not\mathcal{R}t$). A relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}'$ is *total* if for any

$t \in \mathcal{S}$ there is $t' \in \mathcal{S}'$ such that $t\mathcal{R}t'$ (and vice versa) and is *partial* otherwise.

We denote by $\mathcal{R}\circ\mathcal{R}'$ the composition $\{(t,t') \ : \ \exists t''.t\mathcal{R}t'' \wedge t''\mathcal{R}'t'\}$, when it exists. Let Id be the identity relation over any set $\mathcal{S}$. We define the *iterate* $\mathcal{R}^n$ inductively as $\mathcal{R}^0 \stackrel{\text{def}}{=} \mathit{Id}$ and $\mathcal{R}^{n+1} \stackrel{\text{def}}{=} \mathcal{R}\circ\mathcal{R}^n$. We define the reflexive closure as $\mathcal{R}^= \stackrel{\text{def}}{=} \mathit{Id} \cup \mathcal{R}$, the transitive closure as $\mathcal{R}^+ \stackrel{\text{def}}{=} \bigcup_{n\geq 1} \mathcal{R}^n$, the reflexive-transitive closure as $\mathcal{R}^* \stackrel{\text{def}}{=} \mathit{Id} \cup \mathcal{R}^+$, and the *inverse* as $\mathcal{R}^{-1} \stackrel{\text{def}}{=} \{(t',t) \ : \ t\mathcal{R}t'\}$.

A relation is an *equivalence* if it is reflexive, symmetric and transitive. Given two relations $\mathcal{R}, \mathcal{R}'$, if $\mathcal{R} \subset \mathcal{R}'$ we say that $\mathcal{R}$ is *stronger* than $\mathcal{R}'$ and $\mathcal{R}'$ is *weaker* than $\mathcal{R}$.

Functions. A function f with domain $\mathcal{S}$ and codomain $\mathcal{S}'$ is a binary relation over $\mathcal{S}, \mathcal{S}'$ such that whenever $f(t) = t'$ and $f(t) = t''$ it is the case that $t' = t''$. We say that f is *total* if for any $t \in \mathcal{S}$ there is $t' \in \mathcal{S}'$ such that $f(t) = t'$; that it is *injective* if whenever $f(t) = t'$ and $f(t'') = t'$ it is the case that $t = t''$; and that it is *onto* if for any $t' \in \mathcal{S}'$ there is $t \in \mathcal{S}$ such that $f(t) = t'$. A function is *bijective* if it is injective and onto. If f is bijective there exists an *inverse* function f^{-1} such that $f\circ f^{-1} = \mathit{Id}$. We denote by $\mathcal{S} \to \mathcal{S}'$ and $\mathcal{S} \rightharpoonup \mathcal{S}'$ the sets of total and partial functions from $\mathcal{S}$ to $\mathcal{S}'$.

Monoids. A *monoid* $\mathcal{M} = (\mathcal{S}, +, 0, =)$ is a set of terms $t, t', \ldots \in \mathcal{S}$ with a binary operation $+$, a distinguished element 0, and an equivalence relation $=$ such that such that (i) $+$ is associative: $t + (t' + t'') = (t + t') + t''$; (ii) 0 is the neutral element for $+$: $0 + t = t + 0 = t$. If $+$ is also commutative ($t + t' = t' + t$) then we say that $\mathcal{M}$ is a *commutative* monoid.

Chapter 1

Introduction

In this introductory chapter, we set the scene for the whole thesis. In Section 1.1, we describe what dynamic Web data is, we give examples and we explain our motivations for studying this topic. In Section 1.2, we describe the process algebraic approach, by giving an overview of the rest of this document and we summarize our main contributions. In Section 1.3, we discuss the most relevant related work.

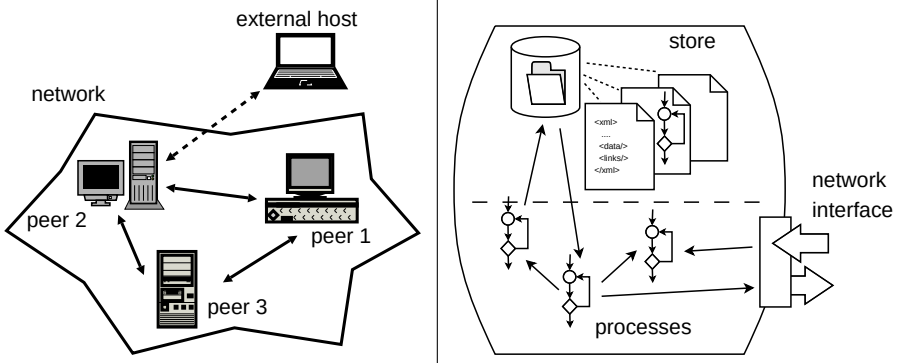
1.1 Dynamic Web data

The World Wide Web is a global network used in daily activities to find information, communicate ideas, conduct business and carry out distributed computations. In order to fully exploit the potential of this massive network, there is a need for scalable mechanisms to organize and manipulate the available information. Peer-to-peer architectures help to deal with the issue of scalability, and technologies such as XML and Web services facilitate the development of distributed applications. XML [81] is a standardized data model which is used to represent uniformly documents containing tagged information which does not adhere to a fixed structure. Web services [83] are Web sites which are designed to be used by applications rather than humans. Web service inter-operability is facilitated by the use of XML for data representation and of related standards for service invocation, description and discovery (SOAP, WSDL, UDDI [84, 82, 76]).

Data integration on the Web constitutes a challenging application for these technologies, because of the extreme heterogeneity of data sources involved, and the complexity of communication patterns which can arise. For example, translating a declarative request for networked data into a low-level execution plan may involve recursively invoking other declarative requests on different Web sites.

Inspired by this problem, in this thesis we study peer-to-peer architectures for exchanging Web data, schematically represented in Figure 1.1. Each *network* is composed by a variable number of interconnected peers, all sharing a similar internal structure, and each one identified by a unique name (Figure 1.1(A)). Peers share a common messaging protocol where the name of a peer is assumed to coincide with its network address: at this level of abstraction there are no restrictions to connectivity due to network domains or firewalls. Networks are *open* in the sense that it

Figure 1.1: Reference architecture



(A) Network. (B) Peer.

is always possible to add new peers or learn dynamically about their existence, and external hosts may participate in the data exchange too, typically playing a limited role. Each peer (schematized in Figure 1.1(B)) acts both as a provider and consumer of information. It contains a data repository, an internal working space where processes carry out local computations, and a network interface providing remote communication and services to other peers. Processes can communicate locally with each other, query and update the local repository and, when the architecture supports mobility, can migrate to other peers to continue execution. Repositories offer to the processes a semi-structured view of their data, which contains enough meta-data to support meaningful queries also in absence of a fixed schema. Data may contain *scripts* or references to other data and services in the form of URLs and queries. A script is some code describing a process which can be interpreted by the working space to add dynamic content to documents. We refer to data of this kind as *dynamic Web data*.

The World Wide Web itself constitutes an extremely general example of architecture for dynamic Web data. Servers use the HTTP protocol to interact with each other, either requesting or providing information. HTML pages can contain hyperlinks, forms and client-side scripts, which provide dynamic behaviour. Web clients running a browser can be considered as the "external hosts" which participate to a smaller degree in the exchange of information.

A more specific example comes from the database world. The Active XML [72, 6] system for data integration (AXML for short) is based on networks of peers each containing a repository of documents and a set of service definitions. AXML Services typically consist of queries and updates on the local repository, but in general can be arbitrary Web services, providing an interface to hosts external to the AXML system. AXML documents are XML documents which can include special tags representing calls to services on other peers. The parameters to these service calls can be local queries (path expressions) or AXML data, hence service calls can be nested. Documents containing service calls are called *intensional documents*, and *materialization* is the process of invoking a service call and pasting its results in the original document. One interesting source of flexibility in AXML is the choice of when to materialize service calls. It can be done periodically, or when the data

containing the call is fetched from the repository, or when it is returned to the client. Similarly, if a service call appears as a parameter to another service call, it can be materialized before calling the service or it can be passed on to it as an intensional parameter.

Besides Web browsers and AXML, a large class of other Web applications (such as file-sharing programs, personal Web portals, online bibliographic databases, etc.) can be seen as instances of the reference architecture given above, each with its own particular features and restrictions. The problems that these architectures have to address, in order to be practically useful, are varied. First of all, it is well-known that interaction between concurrent processes is difficult to regulate. In the case of Web services, this problem is complicated by the difficulty in maintaining state across different Web service invocations, and requires the study of orchestration techniques[1]. Secondly, a major concern for systems dealing with dynamic Web data is security. Depending on the application domain, it may be crucial to have control for example over data integrity, confidentiality, or access control. The formal study of security properties needs to be grounded on a rigourous model of these architectures.

The objective of this thesis is to provide such a formal model of dynamic Web data, and to understand its behavioural properties. We believe that process algebraic techniques are particularly suited for the task at hand, as process algebras have already been successfully used to study concurrent, distributed and mobile systems, and analyze their formal properties, security in particular.

A process-algebraic model could be used for a variety of purposes, such as: derive implementations from abstract specifications using refinement; transparently replace inefficient components with optimized ones; define static analyses (such as types for schema-preservation properties or for security and access control); find bugs in existing systems by analyzing their models. For example, the combination of Web services and scripted processes provides the data engineer with many alternative patterns for exchanging information, and equational reasoning becomes useful to show that some complex data-exchange protocol conforms to its specification.

1.2 A process algebraic approach

Process calculi provide a simple and expressive framework in which to reason about the properties of concurrent, distributed and mobile systems. The π-calculus of Milner, Parrow and Walker [58] is a terse and powerful language which describes the behaviour of concurrent systems, and is endowed with a rich body of theoretical results. It constitutes the basis for many other calculi which target specific aspects of concurrent and distributed systems. Just to mention some of them, the spi-calculus [2] and the applied π-calculus [1] have been used to study security protocols; the distributed π-calculus [43] and Safe Dpi [41] for controlling the access to resources, the Ambient Calculus [20] and the Seal Calculus [79] to study mobile computations across administrative domains; the Join-calculus [26] and Nomadic Pict [77] have been used as a basis for distributed implementations.

[1] By Web service orchestration, we mean a coordination infrastructure which allows modular applications to invoke different Web services and combine their results.

In this thesis, we define the X$d\pi$-calculus with the aim of reasoning about dynamic Web data.

X$d\pi$ terms represent networks of peers where each peer consists of an XML data repository and a working space where π-like processes are allowed to run. We regard processes as agents with a simple set of functionalities: they communicate with each other, query and update the local repository, and migrate to other peers to continue execution. Process descriptions, in the form of scripts, can be included in documents and can be executed by other processes. The definition of X$d\pi$ is parametric with respect to the choice of a specific language of query and update expressions. For the examples, we define a simple such language based on path expressions (Chapter 2).

We tried to keep the definition of X$d\pi$ minimal, including only the basic operations needed to represent the behaviour we are interested in. X$d\pi$ operations consist of asynchronous local communication based on pattern matching, execution of a query-update expression on the local repository, migration, spawning of scripted code and creation of fresh channels. From these, we can derive conditional statements, non-deterministic choices, constructs for parsing and iterating on list-like structures and remote communication in the style of Web services. Using these derived constructs, we can use X$d\pi$ to give a precise semantics to AXML-like behaviour, and consider possible extensions (Chapter 3).

We investigate behavioural equivalences for X$d\pi$. When reasoning about dynamic Web data, the important properties of a network concern what data can eventually be present at a given location. The communication actions of processes, which are the basis of observational equivalences for process calculi, are in principle irrelevant. This shift in perspective is fundamental in our work.

Our network equivalences dictate when two networks can be considered indistinguishable with respect to some externally defined criteria for comparison (henceforth called *observable*). We explore different choices of observables, and we compare the resulting notions of equivalence. In particular, we compare the observable which records the internal structure of documents (which can be easily defined due to the direct representation of data), with the traditional output observable of the asynchronous π-calculus. Network equivalences are parametric with respect to the language used for querying and updating documents (so the generic results are not tied to a particular choice), and can be instantiated to specific cases.

Process equivalence establishes when two processes can replace each other in a network without affecting network equivalence. Ideally, we would like to reason about the equivalence of groups of processes, possibly interacting across several locations, and obtain results which are robust with respect to changes in the data stored in the local repositories and the behaviour of other parallel processes. For these reasons, we introduce a calculus called Core X$d\pi$ which serves as an alternative (semantically equivalent) representation for X$d\pi$ where processes are located explicitly and are separated from the data store. Core X$d\pi$ is suitable for expressing a partial specification of a network by means of located processes running in parallel, possibly sharing private names. Located processes are equivalent if the networks obtained by composing them with arbitrary stores are equivalent. X$d\pi$ networks can be translated into Core X$d\pi$ networks, and the translation preserves network equivalence. X$d\pi$ processes can be mapped to Core X$d\pi$ ones, so that process equivalences

for X$d\pi$ can be studied in Core X$d\pi$ (Chapter 4).

Process equivalences, such as the ones defined in Chapter 4, are hard to use directly because they require a costly property of closure under contexts. Instead, we define a coinductive equivalence relation (called *domain bisimilarity*) which does not quantify over contexts and which entails process equivalence. The definition of domain bisimilarity is non-standard, due to the fact that scripts (which can appear in data) are part of the values, and process equivalences are sensitive to the set of locations constituting the network. We address these two problems by adapting existing techniques for translating messages containing scripts into ones where each script is replaced by a first-order value [69, 46], and by generalizing the notion of bisimulation to families of relations indexed by sets of locations (Chapter 5).

We use bisimilarity to study some communication patterns used by servers in distributed query systems to answer queries from clients. In Core X$d\pi$, distributed queries take the form of processes which retrieve and combine data from different locations by using remote communication and local requests. We show that some existing patterns [68] can be combined together obtaining a flexible infrastructure which is provably equivalent to an intuitive specification of the intended behaviour. By exploiting process migration, we also propose a new communication pattern, and we show that it is behaviourally equivalent to a naive, less efficient one (Chapter 6).

As part of ongoing and future work, we identify two main research directions: the development of a comprehensive type system for X$d\pi$, and the implementation of a platform for exchanging dynamic Web data based on X$d\pi$. A type system, beside ruling out run-time errors due to mismatch in the sorts of operations and their operands, would be useful to study security properties, refine the behavioural equivalences, and guarantee the conformance of data trees to schemas. Preliminary prototype implementations of X$d\pi$ by Imperial College students have given encouraging results about the practical feasibility of our approach (Chapter 7).

Contributions. The non-standard technical challenges of our approach, which incorporates locations, storage and higher order processes in the same framework, can only be fully appreciated by reading through the relevant chapters. Below, we summarize the main contributions of this thesis.

- We define X$d\pi$, a model of peer-to-peer dynamic Web data applications. It extends the asynchronous π-calculus with locations, XML repositories (possibly containing scripts) and query language primitives. We give it a formal semantics, which is parametric in the query and update language chosen.

- We give examples of how X$d\pi$ can be used to model dynamic Web data, in particular comparing our model with AXML.

- We study several definitions of what it means for two X$d\pi$ networks to be equivalent. We consider different choices of observables, define the corresponding network equivalences, and explore the relationships between them.

- We propose Core X$d\pi$ as an equivalent representation for X$d\pi$, where actions are explicitly located and data can be easily separated from processes. We show that the translation from X$d\pi$ to Core X$d\pi$ preserves a large class of network

equivalences, and study process equivalences in Core X$d\pi$. Equivalence of X$d\pi$ processes corresponds to equivalence of their translations in Core X$d\pi$.

- We study a notion of bisimilarity for Core X$d\pi$, by adapting techniques for higher-order processes and introducing a novel treatment of locations. Our equivalence, called *domain bisimilarity*, can be used as a proof method to show equivalence of X$d\pi$ processes or networks.

- We apply our reasoning techniques to show the equivalence of some distributed query patterns.

Our work is one of the first attempts to integrate the study of mobile processes and semi-structured data, and is characterized by its emphasis on dynamic data. It is the first investigation of equivalence properties for (higher-order) data-centric applications based on the Web.

1.3 Related work

The work closest to ours is AXML, which we have described in Section 1.1. Although it has followed an independent development, and is studied by a database (as opposed to our process algebraic) perspective, AXML has been a source of inspiration and comparison to us. On the other hand, despite its expressivity, AXML does not exploit the full potential of dynamic Web data. Service definitions cannot be moved from one peer to another, and service calls are the only kind of data which can give active behaviour to documents. Moreover, service calls have a fixed behaviour: they can only add new results (or replace existing ones) in a specific area of the enclosing document. Thanks to migration, we can express in X$d\pi$ more flexible communication patterns where data can flow to several peers before being returned as a result, possibly to a different peer from the one activating the service call. For example, consider an auditing process for assessing a university course—it goes to a government site, selects the assessment criteria appropriate for the particular course under consideration, then moves this information (a service definition) to the university site to make the assessment, and returns the final result to the auditor. In Chapter 3, we show how to represent and extend AXML-like behaviour in X$d\pi$.

In the remainder of this section, we discuss other related work which does not address dynamic Web data in general, but focuses on some of its specific features (for example distributed queries, service orchestration, or XML). In the remainder of the thesis, we will mention more technical references where appropriate.

1.3.1 Distributed query processing

Distributed query languages extend traditional query languages with facilities for distribution awareness (see Kossmann [51] for a comprehensive survey of the field). They are motivated by the desire to avoid transferring large amounts of data from remote sites in order to answer queries which select only a few results. In general, distributed query systems differ from systems based on dynamic Web data (such as AXML) in that often they focus on a particular database architecture, rely on a

central authority having complete knowledge of the system state, and rarely consider dynamic data. In contrast, the examples described below are nearer to dynamic Web data, and provide potential interesting applications for our framework.

Sahuguet and Tannen propose the ubQL distributed query language [66], which is built by adding process manipulation primitives to any "host" query language. These primitives, inspired by the π-calculus[2], are used in a *deployment phase* to set up a network of processes which, in a successive *execution phase*, will query local repositories and forward their results to other sites, thus implementing a global query execution plan. ubQL processes can deal with streaming data, but there is no support for concurrent execution of query processes on the same site (so in principle the system may not be able to execute more than one global query at a time). The design of X$d\pi$ has been influenced by ubQL, in particular the choice of separating the queries from the process primitives, and maintaining independence from a specific query language. Also our examples on distributed query patterns of Chapter 6 are inspired by ubQL. Overall, the two projects have a significantly different focus and are studied using different methodologies. For example, an important part of the work on ubQL is the study of algorithms for query installation based on cost estimates, which we do not address, whereas behavioural equivalences are not studied in ubQL.

ObjectGlobe [14] is a platform for distributed query evaluation based on an infrastructure of Web servers providing heterogeneous query evaluators and data sources. The idea is that queries involve operators from different query languages suitable for different data formats. A client interacts with a centralized query optimization and meta-data maintenance unit to discover which peers provide operators or data relevant to a specific query, and then dispatches the corresponding subqueries. Hence, ObjectGlobe data is dynamic in the sense that it contains queries to external repositories, which are interpreted on remote sites. The successor project ServiceGlobe [48] shifts the focus from the execution of distributed query plans to that of Web service workflows. Jim and Suciu [47] propose the *dynamically distributed Datalog* (d3log) query language as an extension of Datalog with atoms representing links to other locations and with explicitly located literals (which must be true at the designated location). Such literals are sent as queries to remote sites, and when an answer is found they are sent back to the original site which can resume its resolution process. This simple extension models the discovery of data sources and intensional answers, which are also a distinctive feature of AXML: the answer to a query can contain other located literals. Papadimos and Maier [62] introduce *mutant query plans*, which are XML representations of query plans that can include verbatim XML data, references to resource locations and resource names. They propose a distributed framework where a server receives a mutant query plan, executes part of the plan and then sends what remains to be executed, along with the local results, over to the next server. Kemper and Wiesner [49] propose a scalable architecture for building dynamic virtual market places based on *HyperQueries*, which are query evaluation sub-plans associated to hyperlinks. When a client asks to retrieve a virtual document containing an HyperQuery, the query sub-plan is automatically executed in order to materialize the entire object.

[2]The influence born by the π-calculus on ubQL can be better appreciated considering the preliminary joint work of Sahuguet and Tannen with Pierce [67].

All of these systems are studied from a data-management viewpoint, which our process algebraic techniques could complement nicely. There are many specific issues which are important in databases, such as the use of meta-data to guide the optimization of queries, which we do not study. Instead we have given a formal semantics to the distributed interaction between query processes, arguing about their equivalence and providing a framework on which to base the formal study of security properties.

1.3.2 Web scripting languages

Commercial scripting languages for the Web, such as Javascript, Perl, Python, Php, etc., are widely used to develop Web applications, testifying the feasibility of the scripting approach.

Scripting languages have attracted much interest also in programming language research. For example, Cardelli and Davies [18] propose a set of *Web combinators* providing high-level constructs to mimic the Web surfing behaviour of a human user, on which Kistler and Marais [50] base the WebL scripting language for Web document processing. Graunke *et al.* [38] study the problem of designing interactive Web programs and define the WrForm functional programming language based on an extension of the λ-calculus with *Web forms*, which are records containing a reference to a program and some tagged data. A combination of static and dynamic type safety checks rules out common errors such as trying to access the data field of a form before having instantiated it to a value. Brabrand *et al.* [13] implemented in the <bigwig> project a high-level domain-specific language for programming interactive Web services, geared towards the dynamic generation of Web pages. The core of the language consists of a session-centred service model together with a flexible template-based mechanism for dynamic Web page construction. Static analyses are used to validate the well-formedness of dynamic documents with respect to HTML, and to check concurrency properties using a temporal logics.

With X$d\pi$ we propose a theoretical counterpart to the scripting approach, less domain specific and more focussed on orchestration. An interesting future project could involve the implementation of a realistic scripting language for Web-based data integration based on X$d\pi$ (see Section 7.2).

1.3.3 Process algebras for service orchestration and XML

We conclude our overview of related work by looking at some contributions given by research in process algebras to Web service orchestration and to XML-enabled communication.

Microsoft's BizTalk [23] is a system for orchestrating message-based applications on the Internet which models processes as flowcharts, features short, long, and timed transactions, and provides basic actions for sending or receiving data and controlling the workflow. Bruni *et al.* [16] formalize an operational model of distributed transactions, extending Microsoft BizTalk's short transactions, which can be encoded in the Join calculus. Laneve *et al.* [52] study webπ, an extension of the timed asynchronous π-calculus of Berger and Honda [8, 9] with loosely coupled transactions, which are of interest for Web programming languages. Ferrara [24] gives

a bidirectional translation between the BPEL orchestration language for Web services and the Lotos process algebra. The aim is twofold: to carry out verification at the process algebra level and translate potential counterexamples to BPEL, and also to derive BPEL code directly from algebraic specifications. The representation of compensation handlers (the exception mechanism of BPEL) is simplified by the presence of an explicit disabling operator in Lotos, and is not straightforward in the π-calculus. On the other hand, the encoding of [24] does not include, for example, the dynamic execution of processes, which could be represented quite naturally in the π-calculus thanks to the ability to represent higher-order computations by sending channel names over channels. It is worth noting that the design of XLANG, a precursor of BPEL, was inspired directly by the π-calculus. The cited references show that a process algebraic approach to service orchestration is viable. Although we do not directly study transactions or encodings of orchestration languages in X$d\pi$, many of our examples use directly the expressiveness and flexibility of the π-calculus as a coordination language.

The only work relating the π-calculus with XML which pre-dates ours is the Iota concurrent XML scripting language of Bierman and Sewell [11], used to program Home Area Networks. Iota is a strongly typed functional language with concurrency primitives inspired by the π-calculus. Although the language has a formal semantics, its behavioural theory has not been studied. The authors present a type system which guarantees that XML documents resulting from computations are well-formed (opening and closing tags match, and elements are properly nested), but does not deal with the conformance of a document to an externally defined structure such as a DTD, a Schema or another XML specification. The programs for Home Area devices written in Iota are all supposed to run on the same Home Area server, and the communication with physical devices is modelled through input and output on special channels: distribution is not represented explicitly. Moreover, as opposed to X$d\pi$, the application domain of Home Area Network programming is more control-oriented than data-oriented: there is no explicit representation of stores, which are central to our approach.

Brown *et al.* [15] have recently defined an (untyped) extension of the π-calculus with native XML datatypes called πDuce. They compare its expressivity to that of the functional language XDuce [45], and also consider a higher-order extension which enables dynamic content in documents. An interesting idea underlying the design of πDuce is that processes and data share a similar tree-like structure, and can inhabit the same semantic universe. The authors show a very simple encoding of an evaluator for the subset of the language without new name generation, into the language itself: the execution of processes represented as nested document elements can be simulated in the language. We believe that a similar approach could be taken in X$d\pi$ to represent scripts as semi-structured data, but we prefer hiding the internal structure of processes from the queries, because otherwise it would not be possible to replace a process by an equivalent one whilst preserving the observable behaviour of the system as a whole.

In order to extend XDuce with higher-order functions and a richer algebra of data-types, Benzanken, Castagna *et al.* [7] introduce a non-trivial domain theoretic construction to define the language $\mathbb{C}$Duce. Castagna *et al.* [21] apply the same

approach to define $\mathbb{C}\pi$, a π-calculus extended with pattern matching and tuples of values (XML values can be represented through an encoding). The language comes with a type system featuring intersection and input-output types, achieving a degree of expressivity similar to that of $\mathbb{C}$Duce.

Finally, Acciai and Boreale [4] propose XPi, an extension of the asynchronous π-calculus with code mobility and ML-like pattern matching of structured values. A combination of static and dynamic typing ensures that each channel always exchanges values of the same types. Types can describe the partial structure of a document (hence are more expressive than those in Iota), but do not feature intersection types (hence are less expressive than those in $\mathbb{C}\pi$). X$d\pi$ pattern matching could be easily extended to the more expressive form adopted in $\mathbb{C}\pi$ or XPi. Since in our language query expressions, which are separate entities from processes, are the primary means to extract information from XML trees, we prefer to stick to a simpler definition of pattern matching.

Chapter 2

The X$d\pi$-calculus

In this chapter, we present the Xdπ calculus. In Section 2.1, we introduce Xdπ and explain our design choices through simple examples. In Section 2.2, we define formally the syntax and semantics of Xdπ. In Section 2.3, we define a simple query and update language to be used in concrete examples.

2.1 X$d\pi$ informally

Let us have another look at the diagram of our reference architecture for dynamic Web data given in Chapter 1, reported for convenience in Figure 2.1.

We model each peer as a *location* with a unique name corresponding to the peer identity (for example its IP address). A whole peer-to-peer system is modelled by the parallel composition of the locations corresponding to its peers, which we call a *network*. The XML data stored at each peer is represented by a labelled tree, abstracting away from low-level details. Both the network interface, the services and the working space of each peer are represented by a parallel composition of processes in the corresponding locations. The interface between the working space and the data store is modelled by an operation which the processes of a peer can use for updating or querying the local tree. It is important to clarify that, from now on, we use the word "queries" to mean expressions used to query *or update* a tree. Communication between locations is modelled through process migration, providing a flexible abstraction to model complex coordination protocols.

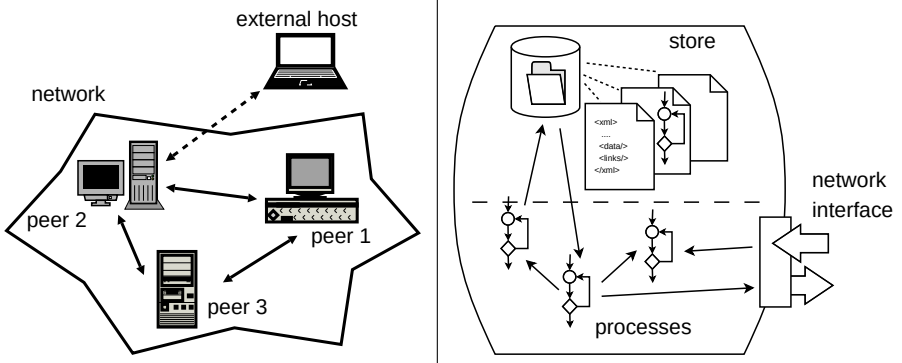
We discuss below the most relevant design choices underlying our models of trees, processes and locations.

2.1.1 The data model

The building blocks of our data model are an abstraction of XML documents (trees), references to data in other documents (pointers) and active components (scripts).

Trees. We use ordered edge-labelled trees to describe XML data. Semi-structured data models are often unordered [3], in contrast with the ordered trees of XML documents. In previous work [34], we considered unordered trees. Here we prefer an ordered model because it is nearer to XML, and has a straightforward correspondence

Figure 2.1: Reference architecture



(A) Network. (B) Peer.

to the textual syntax. The choice of using edge-labelled rather than node-labelled trees is merely a matter of style.

Following a common practice, we do not represent attributes explicitly, but we model them as edges labelled with the attribute name followed by a leaf containing the attribute value. We also embed pointers and scripts as leaves, which in a concrete document are likely to be represented by attributes. The ideas in this thesis do not depend on these particular representation choices. To keep the model simple, we do not represent data values and XML-specific details such as namespaces, ids and idrefs. The tree structure, along with scripts and pointers, provides a sufficiently accurate model for our purposes.

The example in Figure 2.2 (A) shows a fragment of an XML document, and (B) shows its representation in X$d\pi$ (the translation of the hyperlink and the service call are explained below).

Pointers. Hyperlinks have been one of the main features responsible for the success of the Web. We abstract the concept of hyperlink into that of *pointer*, a pair consisting of a location name and a query to identify some data in the tree of the named location. For example, in Figure 2.2 we have translated the hyperlink into a pointer of the form *query*@*location* using the host name "xdpi.net" as the location name, and the path relative to the host "papers/xdpi.pdf" as the query.

Pointers are declarative references which can be interpreted uniformly across locations. A pointer does not specify what to do with the data denoted by the associated query (or update), but typically a process will read the location name and the query from a pointer in order to retrieve some data necessary to continue its execution. Clicking on an HTML hyperlink is a simple example, where the browser process reads the contents of the `href` attribute, retrieves the referenced data, and displays it in the browser window. We can expect that the same query makes sense on different locations, possibly giving different results, because we are assuming that all the peers export their data in the same semi-structured format.

Scripts. Current Web technology is familiar with the use of scripts to provide Web pages with dynamic behaviour. Similarly, we propose to use scripts as a generalization of embedded service calls in the context of Web data integration. Since our scripts

Figure 2.2: Representing XML in X$d\pi$

```
...
⟨data⟩
  ⟨a href = "http://xdpi.net/papers/xdpi.pdf"⟩  Download  ⟨/a⟩
  ⟨call⟩  xdpi.net/getRefs(bibtex)  ⟨/call⟩
⟨/data⟩
...
```

(A) A hyperlink and a service call in XHTML.

```
...
data[
  a[ href[ papers/xdpi.pdf@xdpi.net ]|Download ]
  call[ ⟨go xdpi.net . getRefs⟨bibtex⟩⟩ ]
]
...
```

(B) The translation in X$d\pi$.

are used also for coordination, they are written in the same process language used to describe processes in the working space.

A script is a static piece of code, with some clearly marked parameters and with no reference to the global state except for names of locations and services, which are constants with a uniform meaning across the network. For example, the service call embedded in (A) in Figure 2.2 could be (naively) translated to the script shown in (B), which specifies that a process should go to the host "xdpi.net" and invoke the service "getRefs" with parameter "bibtex". We shall see a more realistic representation of service calls in X$d\pi$ in Chapter 3.

We assume that the representation of scripts as pieces of data is *opaque* (for example, by consisting of a string of bytes which will be interpreted by the system at the time of executing the script) and cannot be parsed or tampered with at run-time, by other processes.[1]

2.1.2 Processes and networks

X$d\pi$ processes are based on the asynchronous π-calculus. To these core processes, we add a migration operation, an update operation for interacting with local data, and a simple form of pattern matching. Values are complex: they can be trees, pointers and scripts.

[1]Some languages such as MetaOCaml [74] and TemplateHaskell [73] provide constructs for multi-stage programming, where pieces of code (possibly containing free variables) can be combined together at run time to form bigger programs, and can be executed. If desired, it is possible to support multi-stage programming in X$d\pi$, defining an XML-like meta-syntax for scripts and interpreting it explicitly using parsing processes in the working space.

Asynchronous communication. The output of a message v on channel a is represented by the asynchronous process $\overline{a}\langle v\rangle$. Since the process has no continuation, the sender cannot be aware of whether a message has been received unless the receiver sends an explicit acknowledgment back.

Asynchronous communication is easy to implement, is natural in a context where failures may occur, and supports many interesting optimizations. For example, in an asynchronous setting, the behaviour of a process cannot be affected by the presence of a communication buffer, a property which in a distributed setting helps to implement location independence.

Migration. Communication across locations is modelled by process migration, which we represent explicitly: the process $\mathsf{go}\, l.P$ represents a (higher-order) message addressed to l containing a request to run the (closed) code P. Due to the peer-to-peer nature of our domain, each location is ready to receive and run any incoming code, so we do not need to provide an explicit operation to run a received process. In some cases, it may also be desirable to give control to each location regarding which code to accept and which to refuse. We leave that task to an eventual superimposed security infrastructure. Using an asynchronous form of communication offers a simple way to model failures within the system. The success of a migration step just depends on the existence of location l. In contrast, the migration rules for other mobile calculi (for example $d\pi$ [42]) assume that migration is always possible. Our choice has an important effect on the behavioural equivalences studied in Section 4.4.

Migration is sometimes criticized for not being efficient enough for practical purposes, as opposed to more basic forms of remote communication. We do not suggest to interpret migration as a recommendation on how to implement communication across locations, but rather as a conceptual device to distinguish clearly local from non-local interaction, useful to express advanced protocols involving code mobility.

Interaction with local data. X$d\pi$ processes access the local tree by using a request operation $\mathsf{req}_p\langle c\rangle$ parametric in a query-update expression p and a channel c. The effect of evaluating expression p is to modify the local tree and to return a list of query results on the specified channel.

Research on query and update languages for XML is still very active [61], and an in-depth study goes beyond the scope of this thesis. Therefore, rather than committing to any particular choice, we parameterize our definitions with respect to an arbitrary query language. Our request operation is defined for any query which given a tree returns an updated tree and a list of results. In Chapter 4, whilst studying equivalences, we will identify some natural conditions on query languages under which equational reasoning becomes more feasible. In Section 2.3, we define explicitly a sample query language, which is just expressive enough to be used in the concrete examples in the rest of the thesis, and which satisfies those simple conditions.

The semantics of the request command specifies that all the results of a query are returned together as siblings in a tree, using a single communication step. We will see in Section 3.1 how this choice is general enough to encode other ways of returning results, such as one result at a time.

Complex values. X$d\pi$ values can be channel names, location names, queries, scripts and trees.

Channel names are partitioned into *private* and *service* channel names. The private channels denote "usual" π-calculus channels, which are typically used for coordination, and which can be kept secret in order to protect a protocol from external interferences[2]. The service channels denote those channels which are used to implement the services which a peer offers to other peers, and which therefore are not meant to be restricted and can be referenced inside scripts.

Both trees and pointers are structured values, which processes need to parse. To this end, we add to π-calculus communication a very simple form of pattern matching. Patterns are terms containing distinct variables which are instantiated, if pattern matching succeeds, with the values found in the corresponding position in the term to be matched. Our patterns do not include regular or recursive expressions, and we will avoid algorithmic issues by simply requiring the guessing of an appropriate substitution in order for pattern matching to take place. Pattern matching for XML-like data is an active research topic, which is orthogonal to our concerns. We believe that the specialized techniques studied elsewhere can be adapted to our setting. Our processes use patterns to parse data, and queries to query trees. This conceptual separation does not exclude the possibility for the query language to be based on pattern matching itself.

Communication in X$d\pi$ is higher-order, in the sense that processes may send scripts over channels, possibly as leaves inside trees. Using a standard "application" command we can pass parameters to a script and run it in the working space.

Locations. An X$d\pi$ network represents a peer-to-peer system, where each location corresponds to a peer. Each peer can communicate with any other peer, and has a unique name. Similarly, our locations are uniquely named and are arranged in a flat domain. The creation of new peers is not an operation which can be performed from within a system, and therefore we do not provide an operation to create new locations. Nevertheless, we will be able to carry on compositional reasoning, hence analyze networks with respect to arbitrary additions of peers. We use the process orchestration techniques associated with the π-calculus to coordinate the movement of data and processes between locations.

2.2 Syntax and semantics

2.2.1 Trees, data and queries

We represent semi-structured data using ordered labelled trees[3]. The formal definition is given in Figure 2.3. We use italic bold letters for arbitrary terms (e.g. $\boldsymbol{t}$) and plain italic letters for closed terms (e.g. t).

Trees and data. We represent a tree as a $\varnothing$-terminated list of branches $E_1 | \ldots | E_n | \varnothing$ which start from the root. Each branch E_i has the form $\mathtt{a}[\,V\,]$ and denotes an edge labelled $\mathtt{a}$ leading to a node containing the data V. A data item can be a subtree T,

[2]We assume that names created using the restriction operator are internal references to the state of a protocol which cannot be guessed from an outsider (say a sufficiently long random bit string).

[3]Our representation of trees is explicit and does not go through a hypothetical encoding into processes, which could be interesting in itself, but would introduce unnecessary complexity, making it hard to reason directly on trees in the equivalences of Chapter 4.

Figure 2.3: Syntax: trees and data

$\boldsymbol{T}, \boldsymbol{S}$ ::=		tree terms
	$\boldsymbol{E}\mathsf{|}\boldsymbol{T}$	branch $\boldsymbol{E}$ composed with tree $\boldsymbol{T}$
	$\varnothing$	empty tree
	x	tree variable
$\boldsymbol{E}, \boldsymbol{F}$ ::=	$\mathtt{a}[\,\boldsymbol{V}\,] \mid x$	branch with edge label $\mathtt{a}$ and data $\boldsymbol{V}$, or variable
$\boldsymbol{U}, \boldsymbol{V}$ ::=		data terms
	$\boldsymbol{T}$	tree $\boldsymbol{T}$
	$\boldsymbol{p}@\boldsymbol{l}$	pointer to location $\boldsymbol{l}$ with query $\boldsymbol{p}$
	$\langle \boldsymbol{A} \rangle$	script $\boldsymbol{A}$
$\boldsymbol{l}$ ::=	$l \mid x$	location name or variable
$\boldsymbol{p}$ ::=	$p \mid x$	query or variable
$\boldsymbol{A}$ ::=	$A \mid x$	script (see Figure 2.4) or variable

$\mathtt{a}, \mathtt{b}, \mathtt{c} \in \mathcal{E}$	(Edge Labels)
$l, m \in \mathcal{L}$ (countably infinite)	(Locations Names)
$p, q \in \mathcal{Q}$	(Queries)
$x, y, z \in \mathcal{V}$	(Variables)
$E, F \in \mathcal{B} \stackrel{\text{def}}{=} \{\boldsymbol{E} \;:\; \mathit{fv}(\boldsymbol{E}) = \emptyset\}$	(Branches)
$U, V \in \mathcal{D} \stackrel{\text{def}}{=} \{\boldsymbol{V} \;:\; \mathit{fv}(\boldsymbol{V}) = \emptyset\}$	(Data)
$T, S \in \mathcal{T} \stackrel{\text{def}}{=} \{\boldsymbol{T} \;:\; \mathit{fv}(\boldsymbol{T}) = \emptyset\}$	(Trees)

Function fv is defined in Figure A.4.
Notation: $\mathtt{a} \stackrel{\text{def}}{=} \mathtt{a}[\,\varnothing\,] \quad E_1\mathsf{|}\ldots\mathsf{|}E_n \stackrel{\text{def}}{=} E_1\mathsf{|}\ldots\mathsf{|}E_n\mathsf{|}\varnothing$.

a pointer $p@l$ referencing the data selected at location l by query p (described below), or a script $\langle A \rangle$ (described in Section 2.2.2) which can be executed to collect data or perform coordination tasks. We show an example of a tree containing a script and a pointer:

$$\mathtt{a}[\,\mathtt{b}[\,\mathtt{c}[\,\langle A \rangle\,]\mathsf{|}\mathtt{d}[\,p@l\,]\,]\mathsf{|}\mathtt{e}\,]$$

We use the same identifiers $x, y, z, \ldots$ to range over all variables. When necessary, the sort of each variable can be understood by the place where the variable occurs. A well sorted substitution is one that for example substitutes a tree term for a tree variable.

Queries. X$d\pi$ is parametric on the choice of a particular query-update language, as long as it is a language of expression which can be evaluated against a tree to obtain some data (the result of querying the tree) and a new tree (the result of updating the tree). The only conditions that we need to impose on such a language are that

the application of a substitution to a query must be well-defined and yield a query[4]. In Chapter 4 we will impose additional conditions required to ensure that a query language is also compatible with our definitions of semantic equivalences.

Definition 2.2.1 (Query Language) *A* query language $(\mathcal{Q}, \mathit{fv}, \mathfrak{E})$ *is a set of queries* $p, q, \ldots \in \mathcal{Q}$ *closed under well-sorted substitutions, together with a function* $\mathit{fv} : \mathcal{Q} \to \wp(\mathcal{V})$ *giving the free variables of each query, and an evaluation function* $\mathfrak{E} : (\mathcal{Q} \times \mathcal{T}) \rightharpoonup \mathcal{T} \times \mathcal{D}^{\prec\omega}$*, which, given a query and a tree, returns an updated tree and a finite list of results.*

Note that in the definition above the evaluation of queries is a partial function. This generality accounts for both the cases of ill-formed queries, which may not have a precise semantics, and Turing-equivalent query languages, which may not terminate. The list of results returned by a query will be reformatted as a tree term when it is passed on to an X$d\pi$ process (see rule (Red Request) in Figure 2.8).

In Section 2.3, we give a concrete query language which will be used in the examples.

2.2.2 Processes and networks

Processes. X$d\pi$ processes extend the processes of the asynchronous π-calculus. The formal definition is given in Figure 2.4. Channel names are partitioned in two disjoint sets: $\mathcal{C}_\mathrm{p}$, containing names of usual π-calculus channels which can appear in the restriction operator $(\nu\, c)\boldsymbol{P}$; and $\mathcal{C}_\mathrm{s}$, containing names of *service channels* which are intended to be known at each location (for example `finger`), and therefore cannot be restricted. The values that can be exchanged over both kind of channels are tuples of channel names, location names, queries, scripts, branches and trees. We assume a simple sorting discipline on variables and channels, to ensure uniformity on the sort and number of values sent along a given channel, and we only consider well-sorted substitutions.

A pattern π is an arbitrary variable x or a data term $\boldsymbol{V}$ which can contain variables (in this case, they must be distinct, by condition $\mathit{distinct}(\boldsymbol{V})$) but cannot contain *complex values* such as scripts and queries (condition $\mathit{cval}(\boldsymbol{V}) = \emptyset$)[5]. Since we are not concerned with algorithmic issues, we define the operation of pattern matching a value v against a pattern π as the guessing of a closing substitution σ such that $v = \pi\sigma$. Since patterns are ordered, if such a substitution exist, it is unique. For notational convenience we will sometimes represent values by their factorization into a pattern and a closing substitution.

An input process $a(\widetilde{\pi}).\boldsymbol{P}$ or $!a(\widetilde{\pi}).\boldsymbol{P}$ is parameterized by a tuple of patterns $\widetilde{\pi}$. All the variables appearing in $\widetilde{\pi}$ are binders with scope $\boldsymbol{P}$, and must be distinct.

[4]The reasons why we need to define substitutions on queries will be clear after describing the semantics of processes in Section 2.2.2.

[5]This condition guarantees that a definition of pattern matching based on syntactic equality will not prevent optimizations based on the semantic equivalences for scripts and queries (which equate more terms). An alternative approach, used for example in Applied-π [1], consists of allowing matching on arbitrary terms, by parameterizing the semantics of pattern matching on an independently defined equivalence relation, which does not have to be syntactic equality. Since our terms include scripts, and the equivalence for scripts is bound to be undecidable, we do not follow this approach.

Figure 2.4: Syntax: X$d\pi$ processes

$\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}$ ::=		process terms
	$\mathbf{0}$	nil process
	$\boldsymbol{P} \mid \boldsymbol{P}$	composition of processes
	$(\nu\, c)\boldsymbol{P}$	private channel c with scope $\boldsymbol{P}$
	$\overline{\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle$	output on service or private channel $\boldsymbol{c}$ of values $\widetilde{\boldsymbol{v}}$
	$\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}$	input on $\boldsymbol{c}$ of $\widetilde{\pi}$ with continuation $\boldsymbol{P}$ ($\mathit{distinct}(\widetilde{\pi})$)
	$!\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}$	lazy replication of an input process ($\mathit{distinct}(\widetilde{\pi})$)
	$\mathsf{go}\ \boldsymbol{l}.\boldsymbol{P}$	migration of $\boldsymbol{P}$ to location $\boldsymbol{l}$
	$\boldsymbol{A} \circ \langle\widetilde{\boldsymbol{v}}\rangle$	run script $\boldsymbol{A}$ with parameters $\boldsymbol{v}$
	$\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle$	request for query $\boldsymbol{p}$ with return channel $\boldsymbol{c}$

$\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ ::= $c \mid \mathsf{c} \mid x$	private (or service) channel name, or variable
$\boldsymbol{v}$ ::= $\boldsymbol{c} \mid \boldsymbol{l} \mid \boldsymbol{p} \mid \boldsymbol{A} \mid \boldsymbol{E} \mid \boldsymbol{T}$	value terms

$a, b, c \in \mathcal{C}_{\mathrm{p}}$ (countably infinite)	(Private Channel Names)
$\mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathcal{C}_{\mathrm{s}}$	(Service Channel Names)
$v, u \in \mathcal{U} \stackrel{\text{def}}{=} \{\boldsymbol{v} \ : \ \mathit{fv}(\boldsymbol{v}) = \emptyset\}$	(Values)
$P, Q, R \in \mathcal{P} \stackrel{\text{def}}{=} \{\boldsymbol{P} \ : \ \mathit{fv}(\boldsymbol{P}) = \emptyset\}$	(Processes)
$\pi \in \mathcal{H} \stackrel{\text{def}}{=} \mathcal{V} \cup \{\boldsymbol{V} \ : \ \mathit{cval}(\boldsymbol{V}) = \emptyset \ \mathit{and}\ \mathit{distinct}(\boldsymbol{V})\}$	(Patterns)
$A \in \mathcal{A} \stackrel{\text{def}}{=} \{(\widetilde{\pi})\boldsymbol{P} \ : \ \mathit{fn}(\boldsymbol{P}) = \emptyset, \mathit{fv}(\boldsymbol{P}) \subseteq \mathit{fv}(\widetilde{\pi}) \ \mathit{and}\ \mathit{distinct}(\widetilde{\pi})\}$	(Scripts)
$C_{\mathcal{P}}[-]$ ::= $- \mid P \mid C_{\mathcal{P}}[-] \mid C_{\mathcal{P}}[-] \mid P \mid (\nu\, c)C_{\mathcal{P}}[-]$	(Process Contexts)

The functions $\mathit{fv}, \mathit{fn}, \mathit{cval}$ and the predicate $\mathit{distinct}$ are defined in Figure A.4 and Figure 2.5.
Notation: $(\nu\, \widetilde{c})\boldsymbol{P} \stackrel{\text{def}}{=} (\nu\, c_1) \ldots (\nu\, c_n)\boldsymbol{P}$; $\overline{\boldsymbol{c}} \stackrel{\text{def}}{=} \overline{\boldsymbol{c}}\langle\varnothing\rangle$; $\boldsymbol{c}.\boldsymbol{P} \stackrel{\text{def}}{=} \boldsymbol{c}(\varnothing).\boldsymbol{P}$.
Convention: we generally omit a trailing $\mathbf{0}$, for example we write $a(x)$ for $a(x).\mathbf{0}$. Sequential composition has syntactic precedence over parallel composition.

For example, process $a(x, \mathsf{a}[\, z@y\,]).\boldsymbol{P}$ is waiting for a pair of values: the first can be arbitrary, the second must be a node labelled a containing a pointer. The intuitive semantics is that, reacting with a matching output such as $\overline{a}\langle b, \mathsf{a}[\, p@l\,]\rangle$, the process will evolve to $\boldsymbol{P}\{{}^{b}/_{x}, {}^{l}/_{y}, {}^{p}/_{z}\}$.

Figure 2.5: Function *cval* and predicate *distinct*

$$\frac{t \in \mathcal{A} \cup \mathcal{Q}}{\mathit{cval}\,(t) = \{t\}} \qquad \mathit{cval}\,(\boldsymbol{E} \mid \boldsymbol{T}) = \mathit{cval}\,(\boldsymbol{E}) \cup \mathit{cval}\,(\boldsymbol{T})$$

$$\mathit{cval}\,(\mathtt{a}[\,\boldsymbol{V}\,]) = \mathit{cval}\,(\boldsymbol{V}) \qquad \mathit{cval}\,(\langle \boldsymbol{A} \rangle) = \mathit{cval}\,(\boldsymbol{A})$$

$$\mathit{cval}\,(\boldsymbol{p}@\boldsymbol{l}) = \mathit{cval}\,(\boldsymbol{p}) \qquad \mathit{cval}\,(\varnothing) = \mathit{cval}\,(l) = \mathit{cval}\,(x) = \emptyset$$

$$\frac{\mathit{distinct}(\pi) \quad \mathit{distinct}(\widetilde{\pi}) \quad \mathit{fv}(\pi) \cap \mathit{fv}(\widetilde{\pi}) = \emptyset}{\mathit{distinct}(\pi, \widetilde{\pi})} \qquad \frac{\mathit{distinct}(\langle \boldsymbol{A} \rangle)}{\mathit{distinct}(\boldsymbol{A})}$$

$$\frac{\mathit{distinct}(\boldsymbol{E}) \quad \mathit{distinct}(\boldsymbol{T}) \quad \mathit{fv}(\boldsymbol{E}) \cap \mathit{fv}(\boldsymbol{T}) = \emptyset}{\mathit{distinct}(\boldsymbol{E} \mid \boldsymbol{T})} \qquad \frac{\mathit{distinct}(\mathtt{a}[\,\boldsymbol{V}\,])}{\mathit{distinct}(\boldsymbol{V})}$$

$$\mathit{distinct}(\varnothing) \qquad \mathit{distinct}(x) \qquad \frac{\mathit{fv}(\boldsymbol{l}) \cap \mathit{fv}(\boldsymbol{p}) = \emptyset}{\mathit{distinct}(\boldsymbol{p}@\boldsymbol{l})}$$

Figure 2.6: Syntax: X$d\pi$ networks

N, M **::=**		networks
	$l\,[\,T \parallel P\,]$	location l containing tree T and process P
	$\mathbf{0}$	empty network
	$N \mid N$	parallel composition of networks (with disjoint domains)
	$(\nu\, c)N$	private channel c with scope N

$N, M \in \mathcal{N}$ (Networks)

$C_{\mathcal{N}}[-]$ **::=** $-$ | $N \mid C_{\mathcal{N}}[-]$ | $C_{\mathcal{N}}[-] \mid N$ | $(\nu\, c)C_{\mathcal{N}}[-]$ (Network Contexts)

The migration primitive $\mathsf{go}\ l.P$ enables a process to go to l and become P.

A script $(\widetilde{\pi})\boldsymbol{P}$ is a process parametric on the patterns $\widetilde{\pi}$ binding in $\boldsymbol{P}$. Scripts can contain service channel names, but cannot contain free occurrences of private channel names (condition $\mathit{fn}(\boldsymbol{P}) = \emptyset$) or free variables (condition $\mathit{fv}(\boldsymbol{P}) \subseteq \mathit{fv}(\widetilde{\pi})$). The application process $\boldsymbol{A} \circ \langle \widetilde{\boldsymbol{v}} \rangle$ starts the execution of the script $\boldsymbol{A}$ passing it the parameters $\widetilde{\boldsymbol{v}}$ (which will be concrete values at run-time).

The request command $\mathsf{req}_{\boldsymbol{p}}\langle \boldsymbol{c} \rangle$ asks the system to perform the update specified by query $\boldsymbol{p}$ on the local tree, and to return the results on channel $\boldsymbol{c}$.

Networks. We model networks as a composition of distinct locations, each containing a tree and a process. The formal definition is given in Figure 2.6.

A location $l\,[\,T \parallel P\,]$ has a unique name l, and contains a tree T and a process P. The empty network is denoted by $\mathbf{0}$. Function $\mathit{dom}(N)$, defined in Figure A.1 returns the *domain* of N (the names of the locations constituting N). Network composition $N \mid M$ is *partial*: the domain of N must be disjoint from that of M. Private channel names can extend their scope across networks, using $(\nu\, c)N$, when a process containing a fresh name migrates to another location. Note that networks

Figure 2.7: Semantics: structural congruence for X$d\pi$

$$(\nu\, c)\mathbf{0} \equiv \mathbf{0} \qquad \text{(Struct Res PNil)}$$

$$c \notin \mathit{fn}(\boldsymbol{P}) \Longrightarrow \boldsymbol{P} \mid (\nu\, c)\boldsymbol{Q} \equiv (\nu\, c)(\boldsymbol{P} \mid \boldsymbol{Q}) \qquad \text{(Struct Res PPar)}$$

$$(\nu\, c)(\nu\, d)\boldsymbol{P} \equiv (\nu\, d)(\nu\, c)\boldsymbol{P} \qquad \text{(Struct Res PRes)}$$

$$(\nu\, c)\mathbf{0} \equiv \mathbf{0} \qquad \text{(Struct Res NNil)}$$

$$c \notin \mathit{fn}(N) \Longrightarrow N \mid (\nu\, c)M \equiv (\nu\, c)(N \mid M) \qquad \text{(Struct Res NPar)}$$

$$(\nu\, c)(\nu\, d)N \equiv (\nu\, d)(\nu\, c)N \qquad \text{(Struct Res NRes)}$$

$$l\,[\,T \parallel (\nu\, c)P\,] \equiv (\nu\, c)l\,[\,T \parallel P\,] \qquad \text{(Struct Res NLoc)}$$

Structural congruence $\equiv$ is a subset of $(\boldsymbol{P} \times \boldsymbol{P}) \cup (\mathcal{N} \times \mathcal{N})$. It is the least equivalence relation satisfying α-conversion and the axioms given above, closed under all the syntactic operators, and such that $(\boldsymbol{P}, \mid, \mathbf{0}, \equiv)$ and $(N, \mid, \mathbf{0}, \equiv)$ are commutative monoids. A complete definition of $\equiv$ can be found in Figure A.3.

do not contain free variables by construction.

2.2.3 Semantics

The semantics of X$d\pi$ is given in terms of structural congruence and a reduction relation, in the style of the Chemical Abstract Machine of Berry and Boudol [10]. The structural congruence for processes and networks is standard, and is defined in Figure 2.7. The reduction relation $\longrightarrow$ describes the movement of processes across locations, the interaction between processes and processes, and the interaction between processes and data. The formal definition is given in Figure 2.8.

Rules (Red Par), (Red Res) and (Red Struct) are standard contextual rules which allow reduction under parallel composition, restriction and structural congruence.

There are two rules for process movement between locations: rule (Red Stay) describes the case where the process is already at the target location, and rule (Red Go) allows a process $\mathtt{go}\ l.P$ to move from m to l.

Rule (Red Com) states that if an output $\overline{a}\langle\widetilde{v}\rangle$ and an input $a(\widetilde{\pi}).\boldsymbol{P}$ on the same channel a are in the same location, and the values $\widetilde{v}$ match the input patterns $\widetilde{\pi}$ (there is a substitution a σ such that $\widetilde{v} = \widetilde{\pi}\sigma$), then communication takes place and execution proceeds with $\boldsymbol{P}\sigma$. Rule (Red Com!) is similar, but leaves the replicated input process $!a(\widetilde{\pi}).\boldsymbol{P}$ in place for further use. We show an example of the communication of a private channel over a service channel below. The reduction step involves the use of structural congruence to extend the scope of the restricted name before communication:

$$l\,[\,T \parallel (\nu\, c)(\overline{\mathtt{a}}\langle c, \mathtt{b}\rangle) \mid \mathtt{a}(x, \mathtt{b}[\,y\,]).(\overline{x}\langle y\rangle \mid \boldsymbol{P})\,] \longrightarrow l\,[\,T \parallel (\nu\, c)(\overline{c}\langle\varnothing\rangle \mid \boldsymbol{P}\{{}^{c}/_{x}, {}^{\varnothing}/_{y}\})\,]$$

Rule (Red Run) runs a script after having pattern matched its parameters. Rule (Red Request) applies the query denoted by $\boldsymbol{p}$ on the local tree T, obtaining an updated

Figure 2.8: Semantics: reduction relation for X$d\pi$

$$l\,[\,T \parallel Q \mid \mathsf{go}\,l.P\,] \;\longrightarrow\; l\,[\,T \parallel Q \mid P\,] \qquad \text{(Red Stay)}$$

$$l\,[\,T \parallel Q \mid \mathsf{go}\,m.P\,] \mid m\,[\,S \parallel R\,] \;\longrightarrow\; l\,[\,T \parallel Q\,] \mid m\,[\,S \parallel R \mid P\,] \qquad \text{(Red Go)}$$

$$l\,[\,T \parallel \overline{c}\langle\widetilde{\pi}\sigma\rangle \mid c(\widetilde{\pi}).\boldsymbol{P} \mid Q\,] \;\longrightarrow\; l\,[\,T \parallel \boldsymbol{P}\sigma \mid Q\,] \qquad \text{(Red Com)}$$

$$l\,[\,T \parallel \overline{c}\langle\widetilde{\pi}\sigma\rangle \mid !c(\widetilde{\pi}).\boldsymbol{P} \mid Q\,] \;\longrightarrow\; l\,[\,T \parallel !c(\widetilde{\pi}).\boldsymbol{P} \mid \boldsymbol{P}\sigma \mid Q\,] \qquad \text{(Red Com!)}$$

$$l\,[\,T \parallel (\widetilde{\pi})\boldsymbol{P} \circ \langle\widetilde{\pi}\sigma\rangle \mid Q\,] \;\longrightarrow\; l\,[\,T \parallel \boldsymbol{P}\sigma \mid Q\,] \qquad \text{(Red Run)}$$

$$\frac{\mathfrak{E}(p,T) = (T', U_1 \mid \ldots \mid U_n \mid \varnothing)}{l\,[\,T \parallel \mathsf{req}_p\langle c\rangle \mid Q\,] \longrightarrow l\,[\,T' \parallel \overline{c}\langle \mathtt{r}[\,U_1\,] \mid \ldots \mid \mathtt{r}[\,U_n\,] \mid \varnothing\rangle \mid Q\,]} \qquad \text{(Red Request)}$$

$$\text{(Red Par)}\ \frac{N \longrightarrow N'}{N \mid M \longrightarrow N' \mid M} \qquad \text{(Red Res)}\ \frac{N \longrightarrow N'}{(\nu\, c)N \longrightarrow (\nu\, c)N'} \qquad \text{(Red Struct)}\ \frac{N \equiv M \longrightarrow M' \equiv N'}{N \longrightarrow N'}$$

Reduction $\longrightarrow$ is a partial relation, subset of $\mathcal{N} \times \mathcal{N}$.
Convention: in this table c ranges over $\mathcal{C}_{\mathrm{p}} \cup \mathcal{C}_{\mathrm{s}}$.

tree T' which replaces T, and a list of results $U_1 \mid \ldots \mid U_n \mid \varnothing$ which is turned into a tree of results (with branches labelled with a standard tag $\mathtt{r}$) sent on channel c. We show a simple example of update, supposing that p is a query which deletes and returns from a tree the data found by following the path $\mathtt{a}/\mathtt{b}$:

$$l\,[\,\mathtt{a}[\,\mathtt{b}[\,V\,] \mid \mathtt{a}[\,U\,]\,] \parallel \mathsf{req}_p\langle c\rangle \mid P\,] \longrightarrow l\,[\,\mathtt{a}[\,\mathtt{b} \mid \mathtt{a}[\,U\,]\,] \parallel \overline{c}\langle \mathtt{r}[\,V\,]\rangle \mid P\,]$$

Note that the subtree V removed from the store is returned as a result by the output on c.

2.3 A sample query and update language

In this section, we define a particular query and update language inspired by XPath [80] which will be used in the examples later on. The result of evaluating a query against a piece of data (when defined) is a pair consisting of a new piece of data, intended to replace the original one, and a list of results, intended to be used by the continuation of the process that executed the query.

The syntax for queries is given in Figure 2.9. Queries are formed by path expressions followed by update expressions. An update expression $(\pi)\boldsymbol{V}$ is a binding pattern followed by a data term. When $(\pi)\boldsymbol{V}$ is applied to a data item U such that $U = \pi\sigma$ for a closing substitution σ, the expression returns the new data item $\boldsymbol{V}\sigma$ and the result U. If there is no such substitution, the expression returns the original data U and the empty result $\varnothing$. A path expression $\mathtt{A}/p$ applied to a tree $\mathtt{a}[\,V\,] \mid T$ evaluates p on V if $\mathtt{a}$ is in the set $\mathtt{A}$, and evaluates itself on the rest of the tree T. A recursive expression $\Box p$ applied to a tree $\mathtt{a}[\,V\,] \mid T$ evaluates p on any node in the tree

Figure 2.9: Syntax: Sam queries

$\widehat{p}, \widehat{q}$ **::=**		path expressions
	ε	empty path
	$\mathtt{A}/\widehat{p}$	follow an edge with label in set $\mathtt{A}$, then $\widehat{p}$
	$\Box\widehat{p}$	follow occurrences of $\widehat{p}$ anywhere
p, q **::=**	$\widehat{p}(\pi)\boldsymbol{V}$	queries

$\mathtt{A}, \mathtt{B} \in \wp(\mathcal{E})$ (Label Sets)
$p, q \in \mathcal{Q}$ (Queries)

Notation: $\mathsf{copy}_{\widehat{p}}(\pi) \stackrel{\text{def}}{=} \widehat{p}(\pi)\pi$; $\mathsf{cut}_{\widehat{p}}(\pi) \stackrel{\text{def}}{=} \widehat{p}(\pi)\varnothing$; $\mathsf{paste}_{\widehat{p}}\langle E\rangle \stackrel{\text{def}}{=} \widehat{p}(x)E\mathsf{I}x$;
$*/\widehat{p} \stackrel{\text{def}}{=} \mathcal{E}/\widehat{p}$; $\mathtt{a}/\widehat{p} \stackrel{\text{def}}{=} \{\mathtt{a}\}/\widehat{p}$.
Convention: we omit a trailing ε from a path, for example we write $\mathtt{A}$ for $\mathtt{A}/\varepsilon$.

in a bottom up fashion[6]. First it evaluates $\Box p$ on V and T, obtaining the updated items V' and T', then it evaluates p on $\mathtt{a}[\,V'\,]\mathsf{I}T'$, combining the results together. The formal definition of query evaluation is given in Figure 2.10.

Definition 2.3.1 (Sample Query Language) *The sample query language* Sam *is the triple* $(\mathcal{Q}, \mathit{fv}, \mathfrak{E})$ *where* $\mathcal{Q}$ *is defined in Figure 2.9,* $\mathfrak{E}$ *is defined in Figure 2.10 and* fv *is defined as* $\mathit{fv}(\widehat{p}(\pi)\boldsymbol{V}) = \mathit{fv}(\boldsymbol{V}) \setminus \mathit{fv}(\pi)$.

Sam is capable of expressing some intuitive tree manipulations. The query $q = \mathsf{copy}_{\mathtt{b}/}(y@x)$ reads the query and the location of any pointer contained in branches labelled $\mathtt{b}$ at the top level. For example,

$$\mathfrak{E}(q, \mathtt{b}[\,T\,]\mathsf{I}\mathtt{b}[\,p@l\,]\mathsf{I}\mathtt{b}[\,q@m\,]) = (\mathtt{b}[\,T\,]\mathsf{I}\mathtt{b}[\,p@l\,]\mathsf{I}\mathtt{b}[\,q@m\,], p@l\mathsf{I}p'@l')$$

The query $q = \mathsf{cut}_{\mathtt{a}/\Box\mathtt{b}/}(x)$ removes the contents of any branch labelled $\mathtt{b}$ found after an initial branch $\mathtt{a}$, and returns the removed data as results. For example,

$$\mathfrak{E}(q, \mathtt{b}[\,V\,]\mathsf{I}\mathtt{a}[\,\mathtt{c}[\,\mathtt{b}[\,U\,]\,]\,]) = (\mathtt{b}[\,V\,]\mathsf{I}\mathtt{a}[\,\mathtt{c}[\,\mathtt{b}\,]\,], U)$$

The query $q = \mathsf{paste}_{\mathtt{a}/*/}\langle\mathtt{e}\rangle$ adds a branch $\mathtt{e}$ to any child of $\mathtt{a}$. For example,

$$\mathfrak{E}(q, \mathtt{a}[\,\mathtt{b}\,\mathsf{I}\mathtt{c}[\,\mathtt{d}\,]\,]) = (\mathtt{a}[\,\mathtt{b}[\,\mathtt{e}\,]\mathsf{I}\mathtt{c}[\,\mathtt{e}\,\mathsf{I}\mathtt{d}\,]\,], \varnothing\mathsf{I}(\mathtt{d}))$$

where the results are the list $\varnothing\mathsf{I}(\mathtt{d})$ where the first element is the empty tree (the contents of $\mathtt{b}$) and the second element is tree $\mathtt{d}$ (the contents of $\mathtt{c}$). Note that the

[6]Suppose we chose a top-down strategy instead. A simple query like "add a subtree $\mathtt{a}$ inside any branch labelled $\mathtt{a}$" on the tree $\mathtt{a}$ should be ruled out, because its evaluation diverges: each time a new subtree is added there is a new branch to update. Inconsistencies of this kind are well-known in languages for updating trees, and there is no general agreement on which strategy should be preferred. Our results do not depend on the strategy chosen for Sam.

Figure 2.10: Semantics: query evaluation for Sam

$$\mathfrak{E}((\pi)\boldsymbol{V}, \pi\sigma) = (\boldsymbol{V}\sigma, \pi\sigma \shortmid \varnothing) \qquad \text{(Eval Match)}$$

$$\mathfrak{E}((\pi)\boldsymbol{V}, U) = (U, \varnothing) \quad \textit{where } U \neq \pi\sigma \qquad \text{(Eval Mismatch)}$$

$$\frac{\mathtt{a} \in \mathtt{A} \quad \mathfrak{E}(p, V) = (V', L) \quad \mathfrak{E}(\mathtt{A}/p, T) = (T', L')}{\mathfrak{E}(\mathtt{A}/p, \mathtt{a}[\,V\,] \shortmid T) = (\mathtt{a}[\,V'\,] \shortmid T', L \shortparallel L')} \qquad \text{(Eval Edge Follow)}$$

$$\frac{\mathtt{a} \notin \mathtt{A} \quad \mathfrak{E}(\mathtt{A}/p, T) = (T', L')}{\mathfrak{E}(\mathtt{A}/p, \mathtt{a}[\,V\,] \shortmid T) = (\mathtt{a}[\,V\,] \shortmid T', L')} \qquad \text{(Eval Edge Discard)}$$

$$\mathfrak{E}(\mathtt{A}/p, U) = (U, \varnothing) \quad \textit{where } U \neq \mathtt{a}[\,V\,] \shortmid T \qquad \text{(Eval Not Edge)}$$

$$\frac{\begin{array}{c}\mathfrak{E}(\Box p, V) = (V', L) \\ \mathfrak{E}(\Box p, T) = (T', L') \\ \mathfrak{E}(p, \mathtt{a}[\,V'\,] \shortmid T') = (T'', L'')\end{array}}{\mathfrak{E}(\Box p, \mathtt{a}[\,V\,] \shortmid T) = (T'', L \shortparallel L' \shortparallel L'')} \qquad \text{(Eval Anywhere Tree)}$$

$$\mathfrak{E}(\Box p, U) = \mathfrak{E}(p, U) \quad \textit{where } U \neq \mathtt{a}[\,V\,] \shortmid T \qquad \text{(Eval Anywhere Else)}$$

Query evaluation $\mathfrak{E}$ is a partial function from $\mathcal{Q} \times \mathcal{D}$ to $\mathcal{D} \times \mathcal{D}^{\prec\omega}$.

query for pasting data is defined only if each selected node (each child of a) contains a tree[7], since otherwise the resulting tree would be ill-formed. The query $q = \Box(\mathtt{b}[\,\langle x \rangle\,] \shortmid y)\mathtt{c}[\,\langle x \rangle\,] \shortmid y$ relabels each branch b containing a script to c:

$$\mathfrak{E}(q, \mathtt{a}[\,\mathtt{b}[\,\langle A \rangle\,] \shortmid \mathtt{b}[\,T\,] \shortmid \mathtt{b}[\,\langle A' \rangle\,]\,]) = (\mathtt{a}[\,\mathtt{c}[\,\langle A \rangle\,] \shortmid \mathtt{b}[\,T\,] \shortmid \mathtt{c}[\,\langle A' \rangle\,]\,], L)$$

where the results $L = (\mathtt{b}[\,\langle A \rangle\,] \shortmid \mathtt{b}[\,T\,] \shortmid \mathtt{c}[\,\langle A' \rangle\,]) \shortmid (\mathtt{b}[\,\langle A' \rangle\,])$ correspond to the two trees to which the pattern was applied successfully. Note that in the first result, which is the last computed, the last branch has already been relabelled.

On the other hand, our sample query language is not sophisticated enough to express (atomically) a query like "delete each branch labelled a which contains a branch labelled b", because if we do not know in advance where a branch labelled b occurs in the contents of a, we cannot write a pattern to select only the nodes containing b. This limitation can be easily overcome by adopting a richer pattern language or by enriching the path expressions with conditions on the contents of nodes (see for example Section 3.3.1).

[7] A type system regulating the contents of trees could prevent processes from getting stuck because of undefined queries.

Chapter 3

Examples

In this chapter, we discuss detailed examples of Xdπ. In Section 3.1, we define some derived constructs which serve as building blocks to express more complex examples. In Section 3.2, we describe our representation of (Web) service calls and service definitions. In Section 3.3, we show how to model the dynamic behaviour of Web data in Xdπ, exploiting links, service calls and scripts in documents.

3.1 Derived constructs

In this section, we define high level commands for conditional expressions, nondeterministic choice, parsing of values, and data manipulation. We give their semantics in terms of a translation into X$d\pi$ primitives. Some of the constructs will be used in the examples later on, some are just for illustrative purposes. We do not claim any particular originality for these definitions, as several ideas are already part of the π-calculus literature or folklore.

To begin with, we introduce a notion of contextual reduction $\stackrel{\circ}{\longrightarrow}$, which describes how a process can evolve independently from the surrounding context. It is a strong property: if a process can evolve into another by a step of $\stackrel{\circ}{\longrightarrow}$, then it has the possibility to do so in any possible context, and it does not need any external resource to do so.

Definition 3.1.1 (Contextual reduction) Contextual reduction *is the smallest relation* $\stackrel{\circ}{\longrightarrow}$ *over processes such that* $P \stackrel{\circ}{\longrightarrow} P'$ *if and only if, for all* $C_{\mathcal{N}}, l, T, C_{\mathcal{P}}$ *such that* $l \notin \mathit{dom}(C_{\mathcal{N}})$, $C_{\mathcal{N}}[l\,[\,T \parallel C_{\mathcal{P}}[P]\,]] \longrightarrow C_{\mathcal{N}}[l\,[\,T \parallel C_{\mathcal{P}}[P']\,]]$.

Conversely, we denote by $P \stackrel{\circ}{\not\longrightarrow}$ the fact that a process cannot reduce, regardless of the surrounding context (*deadlock*). In the examples below, we denote any such process by the constant δ, with the intuitive meaning that it is deadlocked code ready to be garbage-collected.

Definition 3.1.2 (Deadlock) *A process* P *is* deadlocked *if and only if, for all* $C_{\mathcal{N}}, l, T, C_{\mathcal{P}}$ *such that* $l \notin \mathit{dom}(C_{\mathcal{N}})$,

$$C_{\mathcal{N}}[l\,[\,T \parallel C_{\mathcal{P}}[P]\,]] \longrightarrow N \iff C_{\mathcal{N}}[l\,[\,T \parallel C_{\mathcal{P}}[\mathbf{0}]\,]] \longrightarrow N'$$

for arbitrary N, N'. *We abbreviate with* δ *a process* P *such that* $P \stackrel{\circ}{\not\longrightarrow}$.

Figure 3.1: Notation and conventions for the derived constructs

Notation: $\prod_{0\le i\le n} P_i \stackrel{\text{def}}{=} P_1 \mid \ldots \mid P_n$; $\bigoplus_{0\le i\le n} P_i \stackrel{\text{def}}{=} P_1 \oplus \ldots \oplus P_n$; $\sum_{0\le i\le n} P_i \stackrel{\text{def}}{=} P_1 + \ldots + P_n$.

Convention: optional syntactic constructs are enclosed in square brackets $[-]$.

In Figure 3.1, we define some extra notation and conventions used to improve readability. In particular, in the examples below we will use square brackets $[-]$ for optional syntactic constructs.

3.1.1 Control

We begin with some constructs for controlling the execution flow of processes.

If-then-else. The encoding of the if-then-else is the same as in the π-calculus:

(True) $\quad \mathsf{True}(\boldsymbol{c}) \stackrel{\text{def}}{=} \boldsymbol{c}(x,y).\overline{x}$

(False) $\quad \mathsf{False}(\boldsymbol{c}) \stackrel{\text{def}}{=} \boldsymbol{c}(x,y).\overline{y}$

(If Then Else) $\quad \mathsf{if}\ \boldsymbol{c}\ \mathsf{then}\ \boldsymbol{P}[\ \mathsf{else}\ \boldsymbol{Q}] \stackrel{\text{def}}{=} (\nu\, t,f)(\overline{\boldsymbol{c}}\langle t,f\rangle \mid t.\boldsymbol{P}[\mid f.\boldsymbol{Q}]) \quad t,f \notin \mathit{fn}(\boldsymbol{P},\boldsymbol{Q})$

The typical usage consists in initializing a boolean value on a private channel c shared among several processes, and having the processes testing for the value of c, restoring the value (or its negation) after each test. For example, let process $\mathsf{Not}(\boldsymbol{c})$ be the boolean "not" operator on channel $\boldsymbol{c}$:

$$\mathsf{Not}(\boldsymbol{c}) \stackrel{\text{def}}{=} \mathsf{if}\ \boldsymbol{c}\ \mathsf{then}\ \mathsf{False}(\boldsymbol{c})\ \mathsf{else}\ \mathsf{True}(\boldsymbol{c})$$

Let us apply the double negation to the value true on channel c:

$$\begin{aligned}(\nu\, c)(\mathsf{True}(c) \mid \mathsf{Not}(c) \mid \mathsf{Not}(c)) &\stackrel{\circ}{\longrightarrow}\stackrel{\circ}{\longrightarrow}\\ (\nu\, c)(\mathsf{False}(c) \mid (\nu\, f)(f.\mathsf{True}(c)) \mid \mathsf{Not}(c)) &\stackrel{\circ}{\longrightarrow}\stackrel{\circ}{\longrightarrow}\\ (\nu\, c)(\mathsf{True}(c) \mid (\nu\, f)(f.\mathsf{True}(c)) \mid (\nu\, t)(t.\mathsf{False}(c))) &\stackrel{\circ}{\not\longrightarrow}\end{aligned}$$

After the first two reduction steps, channel c carries the boolean value false, and the first negation has become the deadlocked process $(\nu\, f)(f.\mathsf{True}(c))$. After two further reduction steps, the second negation restores the boolean value to true and becomes the deadlocked process $(\nu\, t)(t.\mathsf{False}(c))$.

Internal choice. The intuitive meaning of the internal choice construct $\oplus$ is that process $P \oplus Q$ can only make a $\stackrel{\circ}{\longrightarrow}$ step and become nondeterministically either P or Q. The encoding is straightforward:

(Internal Choice) $\quad \bigoplus_i \boldsymbol{P}_i \stackrel{\text{def}}{=} (\nu\, c)(\overline{c} \mid \prod_i c.\boldsymbol{P}_i) \quad c \notin \mathit{fn}(\boldsymbol{P}_i)$

This mechanism can be easily generalized. For example, to select nondeterministically k processes out of n it is sufficient to use k copies of the output $\overline{c}$. Internal choice can be used for example to represent message loss, both in the case of migration and of local communication:

(Weak Migration) $\underline{\mathsf{go}\,\boldsymbol{l}.\boldsymbol{P}} \stackrel{\text{def}}{=} \mathsf{go}\,\boldsymbol{l}.\boldsymbol{P} \oplus \mathbf{0}$

(Weak Output) $\underline{\overline{\boldsymbol{a}}\langle\widetilde{\boldsymbol{v}}\rangle} \stackrel{\text{def}}{=} \overline{\boldsymbol{a}}\langle\widetilde{\boldsymbol{v}}\rangle \oplus \mathbf{0}$

Indeed, we might replace the normal migration and output constructs with their weak variants to model local or remote links which can be subject to failure.

External choice. We consider now the encoding of external choice, for summations of (possibly replicated) input processes. The intended semantics of external choice is that process $a(x).P + b(y).Q$ can either communicate with an output $\overline{a}\langle v\rangle$ to become $P\{^{v}/_{x}\}$, or with an output $\overline{b}\langle u\rangle$ to become $Q\{^{u}/_{x}\}$, in both cases losing the other input guarded summand. A simple summand can represent an exception handler, which has the power of disabling the other summands when an exception (the corresponding output) occurs. A replicated summand can be a part of a process which is meant to be interruptible. We give the encoding below, assuming the side condition $a, b \notin \mathit{fn}(\boldsymbol{P}) \cup \{c\}$:

(Summand) $\mathsf{branch}(a, \boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{c}(\widetilde{\pi}).\,\mathsf{if}\ a\ \mathsf{then}\ (\boldsymbol{P} \mid \mathsf{False}(a)) \\ \qquad\qquad \mathsf{else}\ (\overline{\boldsymbol{c}}\langle\widetilde{\pi}\rangle \mid \mathsf{False}(a)) \end{cases}$

(!Summand) $\mathsf{branch}(a, !\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) \stackrel{\text{def}}{=} \begin{cases} (\nu\, b)(\overline{b} \mid !b.\boldsymbol{c}(\widetilde{\pi}).\,\mathsf{if}\ a\ \mathsf{then}\ (\boldsymbol{P} \mid \mathsf{True}(a) \mid \overline{b}) \\ \qquad\qquad\qquad\qquad \mathsf{else}\ (\overline{\boldsymbol{c}}\langle\widetilde{\pi}\rangle \mid \mathsf{False}(a)) \end{cases}$

(External Choice) $\sum_i [!]\boldsymbol{c}_i(\widetilde{\omega_i}).\boldsymbol{P}_i \stackrel{\text{def}}{=} (\nu\, a)\big(\mathsf{True}(a) \mid \prod_i \mathsf{branch}(a, [!]\boldsymbol{c}_i(\widetilde{\omega_i}).\boldsymbol{P}_i)\big)$

The (Summand) branch first tries to communicate on channel c, then tests the value of the flag a to check whether or not the choice construct is still active. If a is true, execution can proceed and the flag is set to false (deactivating all the other branches). If a is false, then the output on c is restored and the branch is deadlocked. The case for (!Summand) is similar, but the execution requires the presence of a token b which is restored every time communication is successful. Note that this branch leaves the flag to true, so that the choice construct is still active after communication.

As an example, suppose we have an external choice between an input on channel c and one on channel b. When the corresponding outputs become available, communication on one branch can take place, disabling the other branch. We give sample reductions below:

$$\begin{aligned} & & c(x).P + b(y).Q \mid \overline{c}\langle v\rangle \mid \overline{b}\langle u\rangle \longrightarrow^{\circ} \\ (\nu\, a)\big(\mathsf{True}(a) \mid \left\{\begin{array}{l}\mathsf{if}\ a\ \mathsf{then}\ (P\{^{v}/_{x}\} \mid \mathsf{False}(a)) \\ \mathsf{else}\ (\overline{c}\langle v\rangle \mid \mathsf{False}(a))\end{array}\right\} \mid \mathsf{branch}(a, b(y).Q)\big) \mid \overline{b}\langle u\rangle & \longrightarrow^{\circ}\longrightarrow^{\circ} \\ & & P\{^{v}/_{x}\} \mid (\nu\, a)(\mathsf{False}(a) \mid \delta \mid \mathsf{branch}(a, b(y).Q)) \mid \overline{b}\langle u\rangle \end{aligned}$$

Communication on channel c has happened, and the fact that flag a is false shows

that the other branch of the choice is disabled. In fact,

$$P\{^{v}/_{x}\} \mid (\nu\, a)(\mathsf{False}(a) \mid \delta \mid \mathsf{branch}(a, b(y).Q)) \mid \overline{b}\langle u\rangle \stackrel{\circ}{\longrightarrow}$$
$$P\{^{v}/_{x}\} \mid (\nu\, a)\big(\mathsf{False}(a) \mid \delta \mid \left\{\begin{array}{l}\mathsf{if}\ a\ \mathsf{then}\ (Q\{^{u}/_{y}\} \mid \mathsf{False}(a)) \\ \mathsf{else}\ (\overline{b}\langle u\rangle \mid \mathsf{False}(a))\end{array}\right\}\big) \stackrel{\circ}{\longrightarrow}\stackrel{\circ}{\longrightarrow}$$
$$P\{^{v}/_{x}\} \mid \delta \mid \overline{b}\langle u\rangle$$

The second branch communicates on c, finds that the choice is disabled, restores the output message $\overline{b}\langle u\rangle$ and becomes deadlocked.

3.1.2 Data

We give now some derived constructs which help with data manipulation.

Parsing. Communication in X$d\pi$ is based on pattern matching. By sending a value on a restricted channel, composed in parallel with several inputs with different patterns, we define a nondeterministic pattern matching construct for parsing tuples of values:

(Pattern Matching) $\quad \mathsf{match}\ \widetilde{\boldsymbol{v}}\ \mathsf{with}\ \prod_i \widetilde{\pi}_i\ \mathsf{do}\ \boldsymbol{P}_i$

$$\stackrel{\text{def}}{=}\ (\nu\, c)(\overline{c}\langle\widetilde{\boldsymbol{v}}\rangle \mid \prod_i c(\widetilde{\widetilde{\pi}}_i).\boldsymbol{P}_i) \qquad c \notin \mathit{fn}(\widetilde{\boldsymbol{v}}, \boldsymbol{P}_i)$$

In order to parse all the branches composing the top level of a tree, we can iterate the pattern matching process using replicated inputs, and define an optional base case for the empty tree. Assuming $c \notin \mathit{fn}(\boldsymbol{P}_i, \boldsymbol{Q}), x \notin \mathit{fv}(\pi_i, \boldsymbol{P}_i)$, the code is

(Foreach Tree) $\quad \mathsf{in}\ \boldsymbol{T}\ \mathsf{foreach}\ \prod_i \mathtt{a_i}[\,\pi_i\,]\ \mathsf{do}\ \boldsymbol{P}_i\ [\ \mathsf{ifempty}\ \boldsymbol{Q}]$

$$\stackrel{\text{def}}{=}\ (\nu\, c)(\overline{c}\langle\boldsymbol{T}\rangle \mid \prod_i\, !c(\mathtt{a_i}[\,\pi_i\,]\!\upharpoonright\! x).(\overline{c}\langle x\rangle \mid \boldsymbol{P}_i)[\,\mid c(\varnothing).\boldsymbol{Q}])$$

Below we show an example of parsing a tree value:

$$\mathsf{in}\ \mathtt{a}[\,S\,]\!\upharpoonright\!\mathtt{b}[\,p@l\,]\ \mathsf{foreach}\ \big((\mathtt{a}[\,w\,]\ \mathsf{do}\ \overline{a}\langle w\rangle) \mid (\mathtt{b}[\,y@z\,]\ \mathsf{do}\ \overline{b}\langle z\rangle)\big)\ \ \mathsf{ifempty}\ \overline{c} \stackrel{\circ}{\longrightarrow}$$
$$\mathsf{in}\ \mathtt{b}[\,p@l\,]\ \mathsf{foreach}\ \big((\mathtt{a}[\,w\,]\ \mathsf{do}\ \overline{a}\langle w\rangle) \mid (\mathtt{b}[\,y@z\,]\ \mathsf{do}\ \overline{b}\langle y\rangle)\big)\ \ \mathsf{ifempty}\ \overline{c} \mid \overline{a}\langle S\rangle \stackrel{\circ}{\longrightarrow}$$
$$\mathsf{in}\ \varnothing\ \mathsf{foreach}\ \big((\mathtt{a}[\,w\,]\ \mathsf{do}\ \overline{a}\langle w\rangle) \mid (\mathtt{b}[\,y@z\,]\ \mathsf{do}\ \overline{b}\langle y\rangle)\big)\ \ \mathsf{ifempty}\ \overline{c} \mid \overline{a}\langle S\rangle \mid \overline{b}\langle l\rangle \stackrel{\circ}{\longrightarrow}$$
$$\delta \mid \overline{a}\langle S\rangle \mid \overline{b}\langle l\rangle \mid \overline{c}$$

The code of the for-each loops becomes the deadlocked process

$$\delta = (\nu\, c)(!c(\mathtt{a}[\,w\,]\!\upharpoonright\! x).(\overline{c}\langle x\rangle \mid \overline{a}\langle w\rangle) \mid !c(\mathtt{b}[\,y@z\,]\!\upharpoonright\! x).(\overline{c}\langle x\rangle \mid \overline{b}\langle y\rangle)) \not\stackrel{\circ}{\longrightarrow}$$

which cannot interact anymore with the environment (and in an implementation would ideally be garbage-collected).

Requests. Private channels and pattern matching are also helpful to manipulate the results of requests to the store. A useful construct is the synchronous requests, which binds the result of a request in a continuation process. It can also be combined with an exception handler construct, to model timeouts or other errors which may occur when accessing the store (outputs on the channels e_i). The code is:

(Synchronous Request) $\mathsf{req}_{\boldsymbol{p}}(\pi).\boldsymbol{P}[\ \mathsf{catch}\ \prod_i e_i(\widetilde{\pi}_i)\ \mathsf{do}\ \boldsymbol{Q}_i]$

$$\stackrel{\text{def}}{=}\ (\nu\, c)(\mathsf{req}_{\boldsymbol{p}}\langle c\rangle \mid c(\pi).\boldsymbol{P}[+\sum_i e_i(\widetilde{\pi}_i).\boldsymbol{Q}_i]) \qquad c \notin \mathit{fn}(\boldsymbol{P})$$

Similarly, when the purpose of request is to update the store and not to read data, we can define a synchronous update operation:

(Synchronous Update) $\mathsf{up}_{\boldsymbol{p}}.\boldsymbol{P}[\ \mathsf{catch}\ \prod_i e_i(\widetilde{\pi}_i)\ \mathsf{do}\ \boldsymbol{Q}_i]$

$$\stackrel{\text{def}}{=}\ \mathsf{req}_{\boldsymbol{p}}(x).\boldsymbol{P}[\ \mathsf{catch}\ \prod_i e_i(\widetilde{\pi}_i)\ \mathsf{do}\ \boldsymbol{Q}_i] \qquad c \notin \mathit{fn}(\boldsymbol{P}), x \notin \mathit{fv}(\boldsymbol{P})$$

The synchronous request can be extended along the lines of (Foreach Tree) to parse each result, one by one. Assuming the conditions $c \notin \mathit{fn}(\boldsymbol{P}_i, \boldsymbol{Q}), x \notin \mathit{fv}(\pi_i, \boldsymbol{P}_i)$, the code is given by:

(Foreach Request) $\mathsf{in}\ \boldsymbol{p}\ \mathsf{foreach}\ \prod_i \pi_i\ \mathsf{do}\ \boldsymbol{P}_i\ [\ \mathsf{ifempty}\ \boldsymbol{Q}]\ [\ \mathsf{catch}\ \ldots]$

$$\stackrel{\text{def}}{=}\ \mathsf{req}_{\boldsymbol{p}}(x).(\mathsf{in}\ x\ \mathsf{foreach}\ \prod_i \mathtt{r}[\,\pi_i\,]\ \mathsf{do}\ \boldsymbol{P}_i\ [\ \mathsf{ifempty}\ \boldsymbol{Q}])[\ \mathsf{catch}\ \ldots]$$

For example, we can use (Foreach Request) to codify a run operation which executes all the scripts returned by a request $\boldsymbol{p}$ with some arbitrary parameters $\widetilde{\boldsymbol{v}}$:

(Run) $$\mathsf{run}_{\boldsymbol{p}}\langle\widetilde{\boldsymbol{v}}\rangle \ \stackrel{\text{def}}{=}\ \mathsf{in}\ \boldsymbol{p}\ \mathsf{foreach}\ \langle y\rangle\ \mathsf{do}\ y \circ \langle\widetilde{\boldsymbol{v}}\rangle$$

As an example of the run operation, using the Sam query language, we have

$$T = \mathtt{proc}[\,\langle (x,y)\overline{x}\langle y\rangle\rangle\,]\,|\,\mathtt{b}[\,\mathtt{proc}[\,\langle (x,y)\overline{y}\langle x\rangle\rangle\,]\,]\,|\varnothing$$

$$l\,[\,T \parallel \mathsf{run}_{\mathsf{copy}\,\square}(\langle z\rangle)\langle a,b\rangle\,] \longrightarrow\longrightarrow\longrightarrow\longrightarrow l\,[\,T \parallel \delta \mid \overline{a}\langle b\rangle \mid \overline{b}\langle a\rangle\,]$$

where δ are the deadlocked remains of the for-each loop. In the reductions above, notice that we cannot use the contextual reduction because we have to be explicit about all the network, as the result depends on the actual data in the tree at l.

3.2 Web services

In this section, we describe a simple implementation of macros for defining and calling services in X$d\pi$.

Service definition and service call. A service definition is characterized by the name of the service $\boldsymbol{a}$, its input pattern $\widetilde{\pi}$, its body $(x)\boldsymbol{P}$ and its output pattern $\widetilde{\omega}$. The service receives on channel $\boldsymbol{a}$ the input parameters $\widetilde{\pi}$, a location name y and a channel name z (the latter parameters are used to return the result). The body $(x)\boldsymbol{P}$ takes a fresh channel name c (bound to x) in input, performs some arbitrary computation, and outputs the result on channel c. Note that the variables in $\widetilde{\pi}$ may bind in $\boldsymbol{P}$. A forwarding process inputs the result from channel c according to the output pattern $\widetilde{\omega}$, and forwards it to location y on channel z. The pattern $\widetilde{\omega}$ can be interpreted in the macro below as specifying the output type of the service.

(Service Definition) $\mathsf{Define}\ \boldsymbol{a}(\widetilde{\pi})\ \mathsf{as}\ (x)\boldsymbol{P}\ \mathsf{output}\ \langle\widetilde{\omega}\rangle$
$\stackrel{\text{def}}{=}\ !\boldsymbol{a}(\widetilde{\pi}, y, z).(\nu\, c)((x)\boldsymbol{P} \circ \langle c\rangle \mid c(\widetilde{\omega}).\mathsf{go}\, y.\overline{z}\langle\widetilde{\omega}\rangle)$

The service call is dual. It specifies the location $\boldsymbol{l}$ from which the service is invoked (and to which it is returned), the location $\boldsymbol{m}$ and the name $\boldsymbol{a}$ of the service, its parameters $\widetilde{\boldsymbol{v}}$, and a continuation process $\boldsymbol{Q}$ with patterns $\widetilde{\pi}$ for parsing the results.

(Service Call) $\boldsymbol{l}{\cdot}\mathsf{Call}\ \boldsymbol{m}{\cdot}\boldsymbol{a}\langle\widetilde{\boldsymbol{v}}\rangle\ \mathsf{return}\ (\widetilde{\pi})\boldsymbol{Q}$
$\stackrel{\text{def}}{=}\ (\nu\, c)(\mathsf{go}\, \boldsymbol{m}.\overline{\boldsymbol{a}}\langle\widetilde{\boldsymbol{v}}, \boldsymbol{l}, c\rangle \mid c(\widetilde{\pi}).\boldsymbol{Q})$

The parameters $\boldsymbol{l}$ and c sent on $\boldsymbol{a}$ are used by the forwarding process in the service definition to return the result to the continuation process $\boldsymbol{Q}$. For example, a service providing querying capabilities on its local store, and the corresponding service call, could be defined respectively as

$$\mathsf{Define}\ \mathtt{query}(x_1)\ \mathsf{as}\ (x)\mathsf{req}_{x_1}\langle x\rangle\ \mathsf{output}\ \langle x_2\rangle$$

$$l{\cdot}\mathsf{Call}\ m{\cdot}\mathtt{query}\langle p\rangle\ \mathsf{return}\ (x)\boldsymbol{Q}$$

The service takes as input a query x_1 and executes the corresponding request on the local store. The forwarding part of the service definition will intercept the request result and send it on c at l, where it is passed on to $\boldsymbol{Q}$ on variable x.

Subscriptions. We can easily generalize service definitions to cover the case of *push services*, which send a stream of results to a client in reply to a single service call. The only difference between the code below and (Service Definition) is the presence of a replicated input in the forwarding process

(Push Service) $\mathsf{Push}\ \boldsymbol{a}(\widetilde{\pi})\ \mathsf{as}\ (x)\boldsymbol{P}\ \mathsf{output}\ \langle\widetilde{\omega}\rangle$
$\stackrel{\text{def}}{=}\ !\boldsymbol{a}(\widetilde{\pi}, y, z).(\nu\, c)((x)\boldsymbol{P} \circ \langle c\rangle \mid !c(\widetilde{\omega}).\mathsf{go}\, y.\overline{z}\langle\widetilde{\omega}\rangle)$

The corresponding service subscription waits for multiple results on channel c:

(Subscription) $\boldsymbol{l}{\cdot}\mathsf{Subscribe}\ \boldsymbol{m}{\cdot}\boldsymbol{a}\langle\widetilde{\boldsymbol{v}}\rangle\ \mathsf{return}\ (\widetilde{\pi})\boldsymbol{Q}$
$\stackrel{\text{def}}{=}\ (\nu\, c)(\mathsf{go}\, \boldsymbol{m}.\overline{\boldsymbol{a}}\langle\widetilde{\boldsymbol{v}}, \boldsymbol{l}, c\rangle \mid !c(\widetilde{\pi}).\boldsymbol{Q})$

If desired, the streamed results received by the client can be combined together using a loop similar to the one for (Foreach Tree).

Result forwarding. In order to have complete control on the return parameters, in certain cases we will bypass the service call code, and use only a migration step followed by a service invocation. For example, let

$$\mathit{Service} = \mathsf{Define}\ \mathtt{query}(x_1)\ \mathsf{as}\ (x)\mathsf{req}_{x_1}\langle x\rangle\ \mathsf{output}\ \langle x_2\rangle$$

and consider the network

$$N = (\nu\, c)\left(\begin{array}{l} l\,[\,T_0 \parallel \mathsf{go}\, m.\overline{\mathtt{query}}\langle(w)w, n, c\rangle\,] \\ \mid\ m\,[\,S \parallel \mathit{Service}\,] \\ \mid\ n\,[\,\varnothing \parallel c(x).\mathsf{req}_{(w)x}\langle c\rangle\,] \end{array}\right)$$

The service invocation migrates to l and triggers the service, passing as return parameters n and c. The local request at m copies the whole tree S and forwards it to c on n:

$$N \longrightarrow^* l\,[\,T_0 \parallel \mathbf{0}\,] \mid (\nu\, c)(m\,[\,S \parallel \mathit{Service} \mid \mathsf{go}\, n.\overline{c}\langle S\rangle\,] \mid n\,[\,\varnothing \parallel c(x).\mathsf{req}_{(w)x}\langle c\rangle\,])$$
$$\longrightarrow^* l\,[\,T_0 \parallel \mathbf{0}\,] \mid m\,[\,S \parallel \mathit{Service}\,] \mid n\,[\,S \parallel (\nu\, c)(\overline{c}\langle\varnothing\rangle)\,]$$

At n the code listening on c receives the result and replaces the local tree. We will follow this strategy in several examples, redirecting the results of a service to a location different from the one that issued the service call.

3.3 Dynamic data

In this section, we give examples specific to dynamic Web data which exploit the presence of pointers and scripts in trees. For illustrative purposes, we allow strings as leaves in the data trees.

3.3.1 Bibliographic database

Our first example illustrates a possible usage of pointers as references to data missing from a stored document. Consider a peer l containing a computer science bibliographic database, and a peer m containing a database of Imperial College employees. To start with, the two databases contain mutual references. In the data tree at l, each branch represents a bibliographical item, containing some minimal information about a publication

$$\begin{array}{l}
\texttt{item}[\,\ldots\,]\mid \\
\texttt{item}\big[\texttt{book}[\,\mathit{Data\ on\ The\ Web}\,]\mid \\
\qquad \texttt{authors}[\,\texttt{name}[\,\mathit{Serge\ Abiteboul}\,]\mid\texttt{name}[\,\mathit{Peter\ Buneman}\,]\mid\texttt{name}[\,\mathit{Dan\ Suciu}\,]\,]\,\big]\mid \\
\texttt{item}\big[\texttt{journal}[\,\mathit{Theoretical\ Computer\ Science}\,]\mid \\
\qquad \texttt{article}[\,\mathit{Modelling\ Dynamic\ Web\ Data}\,]\mid \\
\qquad \texttt{authors}[\,\texttt{link}[\,\texttt{id}[\,j\,]\mid\texttt{pointer}[\,p@m\,]\,]\,]\,\big]\mid \\
\quad \ldots
\end{array}$$

The authors of the article "Modelling Dynamic Web Data" are not represented explicitly, but via an intensional reference: the pointer $p@m$, which we will discuss below. The data tree at location m consists of two subtrees, one for academics and one for students, each containing personal data and the publication record for each individual:

$$\begin{array}{l}
\texttt{academics}\big[\texttt{professor}[\,\ldots\,]\mid \\
\qquad \texttt{lecturer}[\,\texttt{name}[\,\mathit{Philippa\ Gardner}\,]\mid\texttt{publications}[\,T_1\,]\mid\ldots\,]\mid \\
\qquad \ldots\big]\mid \\
\texttt{students}\big[\texttt{phd}[\,\ldots\,]\mid \\
\qquad \texttt{phd}[\,\texttt{publications}[\,T_2\,]\mid\texttt{name}[\,\mathit{Sergio\ Maffeis}\,]\mid\ldots\,]\mid \\
\qquad \ldots\big]
\end{array}$$

We assume that both T_1 and T_2 contain a record of the form

$$\begin{aligned}\texttt{article}\big[&\texttt{link}[\,\texttt{id}[\,\ldots\,]\texttt{|}\texttt{pointer}[\,q@l\,]\,]\texttt{|}\\ &\texttt{title}[\,\textit{Modelling Dynamic Web Data}\,]\texttt{|}\\ &\ldots\big]\end{aligned}$$

where the link $q@l$ points to the list of authors for the specific article on location l.

We now describe the two query expressions p and q. For this example, we consider a query language slightly more expressive than Sam which, similarly to XPath, can select edges based on a condition on their contents[1]. For example, path $\texttt{a}[\texttt{b}/\texttt{c}]$ selects the contents of an edge `a`, if it includes an edge `b` followed by another edge `c`. Query p, used at location l to retrieve the authors of an article from the tree of m, is

$$p = \mathsf{copy}_{\square[\texttt{publications}/\texttt{article}/\texttt{title}/\textit{Modelling Dynamic Web Data}]/\texttt{name}}(x)$$

where the path expression $\square[\ldots]/\texttt{name}$ extracts the name field from any node which includes a list of publications containing an article with title "Modelling Dynamic Web Data". Query q, used at location m to retrieve author data from the tree of l, is

$$q = \mathsf{copy}_{\square[\texttt{article}/\textit{Modelling Dynamic Web Data}]/\texttt{authors}}(x)$$

where the path expression $\square[\ldots]/\texttt{authors}$ extracts the list of authors from the article subtree with the correct title.

In order to read all the links in the repository of a location l, and replace each of them with the corresponding data, we define a process $\mathsf{Materialize}(l, \texttt{edge})$, where label `edge` is used to create the edges in which to store each result:

$$\begin{aligned}\mathsf{Materialize}(l,\texttt{edge}) = {}& \mathsf{in}\ \mathsf{copy}_{\square\texttt{link}/}(x)\ \mathsf{foreach}\ \pi_1\ \mathsf{do}\\ & \mathsf{go}\, x_3.\mathsf{req}_{x_2}(y).\mathsf{go}\, l.\mathsf{in}\ y\ \mathsf{foreach}\ \texttt{r}[\,z\,]\ \mathsf{do}\ \mathsf{up}_{p_1}\ \mathsf{ifempty}\ \mathsf{up}_{p_2}\end{aligned}$$

$$\begin{aligned}\pi_1 &= \texttt{id}[\,x_1\,]\texttt{|}\texttt{pointer}[\,x_2@x_3\,]\\ p_1 &= \square[\texttt{link}/\texttt{id}/x_1](z_1\texttt{|}z_2)z_1\texttt{|}\texttt{edge}[\,z\,]\texttt{|}z_2\\ p_2 &= \square[\texttt{link}/\texttt{id}/x_1](z_1\texttt{|}z_2)z_2\end{aligned}$$

The process reads all the link nodes using query $\mathsf{copy}_{\square\texttt{link}/}(x)$, and parses each one using pattern π_1 to extract the id x_1, the query x_2 and the location x_3. The process then goes to x_3, executes x_2 binding the results to y, returns to l, and writes each result in the tree. The update p_1 identifies the node containing the link with the right id and inserts a new branch $\texttt{edge}[\,z\,]$ for each result as a sibling to the link node z_1. The update p_2, which applies when all the results have been pasted, removes the first branch z_1, which contains the link. Suppose that at location l there is the process $\mathsf{Materialize}(l, \texttt{name})$, and at location m there is $\mathsf{Materialize}(m, \texttt{authors})$. We can start

[1] Although logically simple, we have not included this extension in Sam because it would complicate unduly the definition of the evaluation function.

executing Materialize(l, name), obtaining at location l the tree

item[...]|
item[book[*Data on The Web*]|
authors[name[*Serge Abiteboul*]|name[*Peter Buneman*]|name[*Dan Suciu*]]|
...]|
item[journal[*Theoretical Computer Science*]|
article[*Modelling Dynamic Web Data*]|
authors[name[*Philippa Gardner*]|name[*Sergio Maffeis*]]|
...]|
...

where the names of the authors have replaced the link. We can now execute process Materialize(m, authors), transforming the article subtrees of T_1 and T_2 at m into

article[authors[author[*Philippa Gardner*]|author[*Sergio Maffeis*]]|
title[*Modelling Dynamic Web Data*]|
...]

Intensional results. If we change the order in which we execute the two processes, we face a more complex situation. We start executing Materialize(m, authors) on the original tree at location m, and the article subtrees of T_1 and T_2 become

article[authors[link[id[j]|pointer[p@m]]]|
title[*Modelling Dynamic Web Data*]|
...]

The result returned by materializing q@l is again intensional, in the form of an author edge followed by another link (now local) pointing to the author names contained somewhere else in the same tree. The tree obtained at l by executing Materialize(l, name) instead is the same as in the previous example. In order to completely materialize the tree at m, we need to execute also a process Materialize(m, name) which copies the author names from the local tree to the list of authors, reaching the same final state as the previous example.

The process of obtaining a completely materialized document starting from an intensional one is inherently iterative, and in principle may not terminate or be deterministic (see for example [59] for an analysis of intensionality for AXML). Devising static analyses to guarantee convergence in specific cases is an interesting research topic which goes well beyond the scope of this thesis.

3.3.2 Active XML and beyond

In this section, we present an example of dynamic data inspired by AXML, and we generalize it to express a richer data integration scenario.

We have mentioned in Chapter 1 that AXML is based on peers containing a repository of documents and a set of service definitions. AXML services typically

consist of queries and updates on the local repository, but in general can be arbitrary Web services. AXML documents are semi-structured documents which can contain calls to services on other peers. Service calls are essentially remote procedure calls to the specified service on another peer. When the results are returned to the peer which contained the service call, they are added to the original document, possibly replacing the results from previous invocations, or the original service call.

Service calls in X$d\pi$. A straightforward representation of an AXML service call in X$d\pi$is

$$\texttt{call}[\,\texttt{id}[\,j\,]\,|\texttt{code}[\,\langle \mathsf{SCall}(j,l,\mathtt{s})\rangle\,]\,|Pars\,]$$

where `call` is an edge label denoting the service call, `id` is the representation of an id attribute with unique value j, `code` contains the actual script implementing the call to service s on location l, and $Pars$ is a tree containing the parameters to the service. Script $\mathsf{SCall}(j,l,\mathtt{s})$ below takes as input the name of the current location x and the service parameter y, calls service s on l using the macro (Service Call) of Section 3.2, and adds the service results as siblings of the service call node, identified by j:

$$\begin{aligned}\mathsf{SCall}(j,l,\mathtt{s}) = (x,y)x{\cdot}\mathsf{Call}\ l{\cdot}\mathtt{s}\langle y\rangle\ \mathsf{return}\ (z)\\ \mathsf{in}\ z\ \mathsf{foreach}\ \mathtt{r}[\,w_1\,]\ \mathsf{do}\ \mathsf{up}_{\square(\pi_1)\pi_2}\\ \pi_1 = \texttt{call}[\,\texttt{id}[\,j\,]\,|x_1\,]\,|x_2\\ \pi_2 = \texttt{call}[\,\texttt{id}[\,j\,]\,|x_1\,]\,|\mathtt{r}[\,w_1\,]\,|x_2\end{aligned}$$

Pattern π_1 is used to identify and break in two parts the service call node, and pattern π_2 is used for pasting each result in between the service call and the other results. The corresponding service definition can be any instance of (Service Definition) which takes in input a single tree and which returns results of the form $\mathtt{r}[\,V_1\,]\,|\ldots\mathtt{r}[\,V_n\,]\,|\varnothing$. If the service is supposed to return a stream of results instead of a single list, then we can use the macro (Subscription) instead of (Service Call) in the definition above (and correspondingly the macro (Push Service) instead of (Service Definition) for the service definition).

Intensional parameters. A peer can activate a service call in its repository either when a specific timeout expires[2], or when the data containing the service call is being accessed. Before executing the service call, there can be an optional pre-processing phase in which any parameter containing links or other service calls can be materialized or left intensional, depending on the presence of default evaluation rules or explicit overriding annotations. For example, an `immediate` branch in a service call might override a lazy evaluation policy of the parameters. The same considerations in Section 3.3.1 regarding intensional results also apply to the results of service calls.

3.3.3 Stock quoting service

We now give an example of AXML-like service calls in the context of market data retrieval. Suppose that at location l there is a service `gainers` which takes as input the name of a stock exchange and returns the current time and the names of the ten

[2]As we do not model time explicitly in X$d\pi$, a timeout can be represented as an asynchronous interruption in the style of the examples of Section 3.1.

stocks which have gained most points (in percentage) in the last hour, plus a service call to obtain the next ten results. Each day, a dealer at location m invokes the service hourly, and accumulates all the results. The client repository contains the tree

$$\begin{array}{r}\texttt{historical}\big[\texttt{daily}[\,\texttt{date}[\,\mathit{7/7/05}\,]\mid\ldots\,]\mid\\ \texttt{daily}[\,\texttt{date}[\,\mathit{6/7/05}\,]\mid\ldots\,]\mid\\ \ldots\big]\mid\\ \texttt{today}\big[\texttt{bulls}[\,\mathit{CNY}\mid N_n\mid\ldots\mid N_1\,]\mid\\ \texttt{bulls}[\,\mathit{CL}\mid L_n\mid\ldots L_1\,]\mid\\ \ldots\big]\end{array}$$

where `historical` contains the ten best gainers for each hour of the previous days, and `today` contains the current data up to the last service invocation, grouped by stock exchange. CNY and CL are the service calls for the New York and London exchanges:

$$CNY = \texttt{call}[\,\texttt{id}[\,i\,]\mid\texttt{code}[\,\langle\mathsf{SCall}(i,l,\texttt{gainers})\rangle\,]\mid\texttt{exchange}[\,\mathit{nyse}\,]\,]$$

$$CL = \texttt{call}[\,\texttt{id}[\,j\,]\mid\texttt{code}[\,\langle\mathsf{SCall}(j,l,\texttt{gainers})\rangle\,]\mid\texttt{exchange}[\,\mathit{lse}\,]\,]$$

Each N_i (respectively L_i) contains the results for a specific hour at New York (or London). For example

$$N_1 = \texttt{r}[\,\texttt{time}[\,\mathit{9:30am}\,]\mid\texttt{symbol}[\,\mathit{HZO}\,]\mid\texttt{symbol}[\,\mathit{OO}\,]\mid\ldots\mid \mathit{More}_{N_1}\,]$$

where $HZO, OO, \ldots$ are the acronyms of the top gainer stocks, and subtree More_{N_1} contains a service call which can be used to retrieve the next ten best performing stocks for that market at that hour.

A process *Monitor*, in the working space, is responsible for running the scripts every time a specific timeout, represented by a communication on channel t, occurs[3]. At the end of the trading day, a special timeout e causes the process to remove all the service calls from the results of the day, archive them recording the current date, and terminate. The code for process *Monitor* is

$$\begin{aligned}\mathit{Monitor} &= !t.\mathit{Update} + e(x).\mathit{Archive}\\ \mathit{Update} &= \mathsf{req}_{\mathsf{copy}\Box\texttt{bulls}}(x)(\texttt{call}[\,x\mid\texttt{code}[\,\langle y\rangle\,]\mid z\,]\mid w).y\circ\langle m,z\rangle\\ \mathit{Archive} &= \mathsf{up}_{\Box(\texttt{call}[\,y\,]\mid z)z}.\mathsf{req}_{\Box\texttt{today}(y)\varnothing}(z).\mathsf{up}_{(y)\texttt{historical}[\texttt{date}[x]\mid z]\mid y}\end{aligned}$$

When a message $\bar{t}$ becomes available, process *Update* substitutes in the variable y, and executes, the main service call from each `bulls` element, passing the corresponding stock exchange parameter z. The results $N_{n+1}, L_{n+1}, \ldots$ obtained for the various exchanges are added to the data for the current day, which becomes

$$\begin{array}{r}\texttt{today}\big[\texttt{bulls}[\,\mathit{CNY}\mid N_{n+1}\mid N_n\mid\ldots\mid N_1\,]\mid\\ \texttt{bulls}[\,\mathit{CL}\mid L_{n+1}\mid L_n\mid\ldots L_1\,]\mid\\ \ldots\big]\end{array}$$

[3]Since the execution model of X$d\pi$ is not synchronous, our representation of a timeout event as an output message is a drastic simplification, which is useful for this example.

When the message of end of day $\overline{e}\langle \texttt{date}[\ \textit{8/7/05}\ ]\rangle$ becomes available, process *Archive* removes all the service calls from the repository (both the ones used by *Monitor* and the ones returned by service `bulls`, of the form *More*$_-$) with the first update operation, then cuts the data gathered today from the tree with the request, and finally archives it (along with the current date) as historical data using the last update. The resulting tree at m becomes

$$\begin{array}{l}\texttt{historical}\Big[\texttt{daily}[\ \texttt{date}[\ \textit{8/7/05}\ ]\texttt{|bulls}[\ N'_{n+1}\texttt{|}\ldots\texttt{|}N'_1\ ]\texttt{|bulls}[\ L'_{n+1}\texttt{|}\ldots\texttt{|}L'_1\ ]\texttt{|}\ldots\ ]\texttt{|}\\ \qquad\qquad \texttt{daily}[\ \texttt{date}[\ \textit{7/7/05}\ ]\texttt{|}\ldots\ ]\texttt{|}\\ \qquad\qquad \texttt{daily}[\ \texttt{date}[\ \textit{6/7/05}\ ]\texttt{|}\ldots\ ]\texttt{|}\\ \qquad\qquad \ldots\Big]\texttt{|}\\ \qquad\texttt{today}\end{array}$$

where each N'_i is N_i without the service call for additional results, and similarly for L'_i.

3.3.4 Market indexes

In the previous example, we have used AXML calls as a standard way to add dynamic behaviour to trees. They are more flexible than pointers, as used in Section 3.3.1, in that the service being called can perform arbitrary operations to produce the results, whereas pointers are tied up to a specific query-update. A further step of generalization consists of including arbitrary scripts in data trees. The behaviour that unconstrained scripts can provide is more flexible than just invoking a remote service and handling its results locally: they allow for directly accessing remote data, gathering results from different locations, forwarding them to other agents and invoking other services. Moreover, thanks to the ability to store arbitrary scripts, we can represent documents organized as code libraries, from which scripts can be read and then transferred for execution on other locations.

We now extend the example of Section 3.3.3 by adding a market analysis location n containing a database of market indexing functions which interact directly with raw market data, and are assumed to compute a specific index value by averaging the price of a determined set of stocks. Clients can read these functions from the tree, and migrate to some location providing market data to apply the functions and obtain the index values.

The indexing functions are stored in the tree at n in the branch

$$\begin{array}{l}\texttt{indexes}\Big[\texttt{S\&P500}[\ \texttt{script}[\ \langle (x)\boldsymbol{P}_1\rangle\ ]\ ]\texttt{|}\\ \qquad\qquad \texttt{DowJonesIA}[\ \texttt{script}[\ \langle (x)\boldsymbol{P}_2\rangle\ ]\ ]\texttt{|}\\ \qquad\qquad \texttt{NASDAQC}[\ \texttt{script}[\ \langle (x)\boldsymbol{P}_3\rangle\ ]\ ]\texttt{|}\\ \qquad\qquad \ldots\Big]\end{array}$$

where each $\boldsymbol{P}_i$ is a functional process which takes a private channel name (bound to x) as input, and computes a specific index outputting the result on channel x (at the location where it is executed). We can add to the `today` branch in the tree at

location m (see Section 3.3.3) a script for obtaining the value of the main market indexes. The new tree becomes

$$\begin{array}{l}\texttt{today}\big[\texttt{indexes}[\,\texttt{call}[\,\texttt{id}[\,h\,]\texttt{|code}[\,\langle Ind\rangle\,]\,]\,]\texttt{|}\\ \qquad\texttt{bulls}[\,CNY\texttt{|}N_n\texttt{|}\ldots\texttt{|}N_1\,]\texttt{|}\\ \qquad\texttt{bulls}[\,CL\texttt{|}L_n\texttt{|}\ldots L_1\,]\texttt{|}\\ \qquad\ldots\big]\end{array}$$

where the code of script Ind, given below, is based on the code for $\mathsf{SCall}(-,-,-)$:

$$\begin{aligned}Ind &= (x,y)\mathsf{go}\,n.\mathsf{in}\ \mathsf{copy}_{\square\texttt{indexes}/*/\texttt{code}}(z)\ \mathsf{foreach}\ \langle w\rangle.\\ &\qquad \mathsf{go}\,l.(\nu\,c)(w\circ\langle c\rangle\mid c(z).\mathsf{go}\,x.\mathsf{up}_{\square(\omega_1)\omega_2})\\ \omega_1 &= \texttt{call}[\,\texttt{id}[\,h\,]\texttt{|}x_1\,]\texttt{|}x_2\\ \omega_2 &= \texttt{call}[\,\texttt{id}[\,h\,]\texttt{|}x_1\,]\texttt{|index}[\,z\,]\texttt{|}x_2\end{aligned}$$

The new script has a more sophisticated behaviour than a service call. It takes in input the current location x and the call parameter y (the latter only for uniformity), goes to location n, and reads off the tree all the index functions. For each one of them, a subprocess goes to location l (containing the actual market data), creates a fresh channel c, runs the function, waits for the result on c and goes back to the original location x to paste the result as a sibling of the service call. The result of executing the script is

$$\begin{array}{l}\texttt{today}\big[\texttt{indexes}[\,\texttt{call}[\,\ldots\,]\texttt{|index}[\,I_n\,]\texttt{|}\ldots\texttt{|index}[\,I_1\,]\,]\texttt{|}\\ \qquad\texttt{bulls}[\,CNY\texttt{|}N_n\texttt{|}\ldots\texttt{|}N_1\,]\texttt{|}\\ \qquad\texttt{bulls}[\,CL\texttt{|}L_n\texttt{|}\ldots L_1\,]\texttt{|}\\ \qquad\ldots\big]\end{array}$$

where each I_i is the data concerning a particular index.

Along these lines, it is possible to devise more complex scripts which combine service invocations, local queries, and coordination processes to facilitate data integration. For example, in Chapter 6 we will discuss sophisticated communication patterns which can be adapted to X$d\pi$ and combined with the examples given above.

Chapter 4

Behavioural Equivalences

In this chapter, we study behavioural equivalences for Xdπ networks and processes. In Section 4.1, we define a notion of reduction congruence induced by a set of observables *which constitutes the basis for our behavioural equivalences. In Section 4.2, we study particular instances of reduction congruences targeted at specific properties of Xdπ networks. Process equivalence establishes when two processes can replace each other in a network without affecting network equivalence. In order to define process equivalence, we need to separate reasoning about processes from reasoning about data. To do so, in Section 4.3, we define an alternative notation for Xdπ networks called Core Xdπ, and we show that under some mild assumptions the behavioural equivalences are the same in the two formalisms. Finally, in Section 4.4, we define what it means for Core Xdπ processes to be equivalent, in such a way that when equivalent processes are replaced in the same network, the observable behaviour of the network does not change.*

4.1 Equivalences and observables

In this section, we propose a notion of behavioural equivalence for a reduction system (a set of terms along with a reduction relation) such that two terms are equivalent if they enjoy the same (independently defined) properties, and if they remain equivalent after performing reduction steps or being embedded in larger contexts. We call such an equivalence an *induced reduction congruence*, since its discriminating power depends on the particular properties chosen to compare terms. At the opposite extremes, if the property considered is "being a term" then the equivalence relation is trivial (equating all terms), whereas if the property is "having an infinite reduction sequence" then the equivalence is likely to be undecidable.

4.1.1 Induced reduction congruence

We base our definition of induced reduction congruence on similar definitions widely used in the literature, but we consider a more abstract notion of observation predicate. In fact, we identify a property with the set of terms enjoying it, and we call such set an *observable*. We define an *observation predicate* as the characteristic function of the corresponding observable.

Definition 4.1.1 (Induced Reduction Congruence) *Let $(\mathcal{S}, \longrightarrow)$ be a reduction system, $\mathcal{K} \subseteq \mathcal{S} \to \mathcal{S}$ be a set of contexts, and $\mathcal{O} \subseteq \wp(\mathcal{S})$ be a set of observables.*

Given an observable $\alpha \in \mathcal{O}$ we define the observation predicate $\downarrow_\alpha$ *as $t \downarrow_\alpha \iff t \in \alpha$ and the* weak *observation predicate $\Downarrow_\alpha$ as $t \Downarrow_\alpha \iff \exists t'.t \stackrel{*}{\longrightarrow} t'$ and $t' \downarrow_\alpha$.*

The reduction congruence induced by $(\mathcal{S}, \longrightarrow, \mathcal{K}, \mathcal{O})$*, and denoted by $\simeq_\mathcal{O}$, is the largest symmetric relation on $\mathcal{S} \times \mathcal{S}$ which is*

- *observation preserving: $t \simeq_\mathcal{O} s \Longrightarrow \forall \alpha \in \mathcal{O}.t \Downarrow_\alpha \Longrightarrow s \Downarrow_\alpha$*
- *reduction closed: $t \simeq_\mathcal{O} s \Longrightarrow \forall t' \in \mathcal{S}.t \stackrel{*}{\longrightarrow} t' \Longrightarrow \exists s'.s \stackrel{*}{\longrightarrow} s'$ and $t' \simeq_\mathcal{O} s'$*
- *contextual: $t \simeq_\mathcal{O} s \Longrightarrow \forall C[-] \in \mathcal{K}.C[t] \simeq_\mathcal{O} C[s]$.*

When the observables are not important, or implied by the context, we denote $\simeq_\mathcal{O}$ simply by $\simeq$.

In order to use equational reasoning, it is important to remark that reduction congruence is an equivalence relation.

Observation 4.1.2 (Equivalence) *Any induced reduction congruence $\simeq$ is an equivalence relation.*

Proof. By definition $\simeq$ is symmetric. We show by contradiction that it is also reflexive and transitive. Suppose that $(t,t) \notin \simeq$ and let $\dot{\simeq}$ be defined as $\simeq \cup \{(C[t], C[t]) \ : \ C[-] \in \mathcal{K}\}$. It is easy to see that $\dot{\simeq}$ respects the requirements of Definition 4.1.1, and since it is larger than $\simeq$ we have reached a contradiction. The case for transitivity is analogous. □

In order to compare reduction congruences based on the same reduction system, but induced by different observables, it is sufficient to study whether the observation preserving property of one entails the analogous property on the other. The technical lemma below, which formalizes this property, will be used several times in the rest of this chapter.

Lemma 4.1.3 (Inclusion) *Consider a reduction system $(\mathcal{S}, \longrightarrow)$ with contexts $\mathcal{K}$ and two sets of observables $\mathcal{O}$ and $\mathcal{O}'$. If, for any $\alpha \in \mathcal{O}$ and $(t,s) \in \simeq_{\mathcal{O}'}$*

$$t \downarrow_\alpha \Longrightarrow \exists C[-] \in \mathcal{K}, \beta \in \mathcal{O}'.\, C[t] \Downarrow_\beta \ \text{ and } \ (C[s] \Downarrow_\beta \Longrightarrow s \Downarrow_\alpha)$$

then $\simeq_{\mathcal{O}'}$ is included in $\simeq_\mathcal{O}$.

Proof. By definition, $\simeq_{\mathcal{O}'}$ is symmetric, reduction closed and contextual. All we need to show is that it preserves $\mathcal{O}$, because by definition $\simeq_\mathcal{O}$ is the largest such relation.

Consider an arbitrary pair $(t,s) \in \simeq_{\mathcal{O}'}$. Suppose $t \Downarrow_\alpha$ for $\alpha \in \mathcal{O}$. By definition, there is a t' such that $t \stackrel{*}{\longrightarrow} t'$ and $t' \downarrow_\alpha$. By reduction closure, there is an s' such that $s \stackrel{*}{\longrightarrow} s'$ and $t' \simeq_{\mathcal{O}'} s'$. By hypothesis, there are $C[-]$ and $\beta \in \mathcal{O}'$ such that $C[t'] \Downarrow_\beta$. By contextuality, $C[t'] \simeq_{\mathcal{O}'} C[s']$. Since $\beta \in \mathcal{O}'$ and $\simeq_{\mathcal{O}'}$ preserves $\mathcal{O}'$, then $C[s'] \Downarrow_\beta$.

By hypothesis, $C[s'] \Downarrow_\beta \Longrightarrow s' \Downarrow_\alpha$. By $s \longrightarrow^* s'$ and $s' \Downarrow_\alpha$, we conclude with $s \Downarrow_\alpha$. □

The choice of which observables are to be preserved largely determines whether two terms are equivalent or not. In general, the larger the set of observables taken into consideration, the stronger the discriminating power of the resulting reduction congruence. On the other hand, also considering smaller observables (that is, more specific properties) has a similar strengthening effect.

Lemma 4.1.4 (Observation Power) *Consider a fixed reduction system $(\mathcal{S}, \longrightarrow)$ with two sets of observables $\alpha, \ldots \in \mathcal{O}$ and $\beta, \ldots \in \mathcal{O}'$:*

1. *if $\mathcal{O} \subseteq \mathcal{O}'$ then $\simeq_{\mathcal{O}'} \subseteq \simeq_{\mathcal{O}}$;*
2. *if whenever $t \in \alpha$ there is $\beta \subseteq \alpha$ such that $t \in \beta$, then $\simeq_{\mathcal{O}'} \subseteq \simeq_{\mathcal{O}}$.*

Proof. Both cases follow from Lemma 4.1.3 noticing that $\simeq_{\mathcal{O}'}$ preserves $\mathcal{O}$. Point 1 uses the fact that any observable α in $\mathcal{O}$ is also in $\mathcal{O}'$. Point 2 uses the fact that by definition $t \downarrow_\beta \Longrightarrow t \downarrow_\alpha$. □

If a reduction congruence is based on contexts which do not inhibit reduction, then a simple criterium for equivalence (independent from the choice of observables) consists in checking if two terms can reduce to each other.

Definition 4.1.5 (Reduction Contexts) *Given a reduction system $(\mathcal{S}, \longrightarrow)$, we say that $\mathcal{K} \subseteq \mathcal{S} \to \mathcal{S}$ is a set of* reduction contexts *if for any $t \in \mathcal{S}$, $C[-] \in \mathcal{K}$, whenever $t \longrightarrow^* t'$ then $C[t] \longrightarrow^* C[t']$.*

Lemma 4.1.6 (Mutual Reduction) *Let $\simeq$ be the reduction congruence induced by some arbitrary $(\mathcal{S}, \longrightarrow, \mathcal{K}, \mathcal{O})$, where $\mathcal{K}$ is a set of reduction contexts, $\mathcal{O}$ is an arbitrary set of observables, and $t, s \in \mathcal{S}$. If $t \longrightarrow^* s$ and $s \longrightarrow^* t$ then $t \simeq s$.*

Proof. We show that the relation

$$\dot{\simeq} = \left\{ (t, s) \;\; : \;\; t \longrightarrow^* s,\, s \longrightarrow^* t \right\}$$

is contained in $\simeq$. By Definition 4.1.1, we need to show that $\dot{\simeq}$ is symmetric, observation preserving, reduction closed and contextual. The definition of $\dot{\simeq}$ is clearly symmetric. We show that $\dot{\simeq}$ preserves observations. Consider an arbitrary pair $(t, s) \in \dot{\simeq}$. Suppose $t \Downarrow_\alpha$ for$\alpha \in \mathcal{O}$. It must be the case that $t \longrightarrow^* t' \downarrow_\alpha$. By hypothesis $s \longrightarrow^* t$, hence $s \Downarrow_\alpha$. We show that $\dot{\simeq}$ is reduction closed. Suppose $t \longrightarrow^* t'$. By hypothesis $s \longrightarrow^* t$, hence $s \longrightarrow^* t'$ and trivially $t' \dot{\simeq} t'$. We show that $\dot{\simeq}$ is contextual. We need to show that, for an arbitrary $C[-]$, $C[t] \dot{\simeq} C[s]$. Since $C[-]$ is a reduction context, $t \longrightarrow^* s \Longrightarrow C[t] \longrightarrow^* C[s]$ and $s \longrightarrow^* t \Longrightarrow C[s] \longrightarrow^* C[t]$. By definition, $C[t] \dot{\simeq} C[s]$. □

4.1.2 Limit observables

We conclude this section by considering some extreme sets of observables: that is, those for which the induced reduction congruence is the largest or the smallest. In Section 4.2.1, we will consider sets of observables which are of specific interest for X$d\pi$.

Definition 4.1.7 (Limit Observables) *Let $(\mathcal{S}, \longrightarrow)$ be an arbitrary reduction system with contexts $\mathcal{K}$. The* top observables *are the singleton $\mathcal{O}_\top \stackrel{\text{def}}{=} \{\mathcal{S}\}$. The* bottom observables *are the singleton $\mathcal{O}_\bot \stackrel{\text{def}}{=} \{\emptyset\}$. The* identity observables *are the set $\mathcal{O}_= \stackrel{\text{def}}{=} \{\{t\} \ : \ t \in \mathcal{S}\}$. We denote by $\simeq_\top$ the reduction congruence induced by $(\mathcal{S}, \longrightarrow, \mathcal{K}, \mathcal{O}_\top)$, and similarly for the other observables.*

Besides the sets of observables defined above, we know that by definition the smallest set of observables is $\emptyset$, and the largest is $\wp(\mathcal{S})$.

Proposition 4.1.8 (Limit Congruences) *The largest induced reduction congruence $\mathcal{S} \times \mathcal{S}$ coincides with $\simeq_\emptyset$, $\simeq_\top$ and $\simeq_\bot$. The smallest induced reduction congruence coincides with $\simeq_{\wp(\mathcal{S})}$ and $\simeq_=$.*

Proof. The subsets of $\wp(\mathcal{S})$ ordered by inclusion form a complete lattice with greatest element $\wp(\mathcal{S})$ and smallest element $\emptyset$. By point 1 of Lemma 4.1.4, $\simeq_\emptyset$ is the largest induced reduction congruence, and $\simeq_{\wp(\mathcal{S})}$ is the smallest. Both $\simeq_\top$ and $\simeq_\bot$ denote the same total relation as $\simeq_\emptyset$, since they trivially respect the observables: $t \Downarrow_\top$ for all t, and there is no t such that $t \Downarrow_\bot$. Let $\mathcal{O}' = \wp(\mathcal{S}) \setminus \mathcal{O}_\bot$. By point 2 of Lemma 4.1.4, $\simeq_= \subseteq \simeq_{\mathcal{O}'}$. Since there is no t such that $t \Downarrow_\bot$, $\simeq_{\mathcal{O}'} = \simeq_{\wp(\mathcal{S})}$. Since $\simeq_{\wp(\mathcal{S})}$ is the smallest induced reduction congruence, $\simeq_= = \simeq_{\wp(\mathcal{S})}$. □

4.2 Network equivalences

In this section, we define reduction congruences where the reduction system is $(\mathcal{N}, \longrightarrow)$ and the contexts $\mathcal{K}_\mathcal{N}$ are the reduction contexts for networks defined in Figure 2.6. We consider different choices of observables, and we study the formal relationship between the reduction congruences induced by the ones which we consider most interesting.

4.2.1 X$d\pi$ observables

Quite interestingly, in X$d\pi$ and in other calculi with a semantics based on reduction and structural congruence, the minimal induced reduction congruence $\simeq_=$ does not coincide with syntactic equality, nor with structural congruence.

Observation 4.2.1 (Minimal Reduction Congruence for X$d\pi$) *In the X$d\pi$ calculus, $= \subsetneq \simeq_= \subsetneq \equiv$.*

Proof. The first strict inclusion follows from Observation 4.1.2 and the counterexample

$$l\,[\,\varnothing \parallel (\nu\, c)(\overline{c}\,|\,!c.\overline{c})\,] \simeq_{=} l\,[\,\varnothing \parallel (\nu\, c)(!c.\overline{c}\,|\,\overline{c})\,]$$

where the two networks can reduce to each other using (Red Struct) with premise (Red Com!), and are therefore equivalent by Lemma 4.1.6. The second strict inclusion follows from Proposition 4.1.8, noticing that $\equiv$ coincides with the reduction congruence induced by observing networks up to structural congruence, and the counterexample

$$l\,[\,\varnothing \parallel \mathbf{0}\,] \,|\, \mathbf{0} \not\simeq_{=} \mathbf{0} \,|\, l\,[\,\varnothing \parallel \mathbf{0}\,]$$

where the two networks have different observables and cannot reduce. □

In view of Observation 4.2.1, we find it useful to define the class of observables for which structural congruence is the smallest induced reduction congruence.

Definition 4.2.2 (Observables up-to $\equiv$) *A set of observables $\alpha, \ldots \in \mathcal{O}$ is defined* up-to structural congruence *if whenever $N \downarrow_\alpha$ and $N \equiv N'$ then $N' \downarrow_\alpha$.*

From now on, we consider only sets of observables defined up-to structural congruence. In particular, we focus on X$d\pi$-specific observables based respectively on properties of locations, trees and processes.

Location observables. We start with the location observables, which characterize the domain of a network: two networks are equivalent if and only if they have the same domain.

Definition 4.2.3 (Location Observables) *Let a* location observable $l[\,]$ *denote the set* $l[\,] \stackrel{\text{def}}{=} \{N \;:\; l \in \mathit{dom}(N)\}$. *The* location observables *are the set* $\mathcal{O}_{\mathrm{d}} \stackrel{\text{def}}{=} \{l[\,] \;:\; l \in \mathcal{L}\}$.

Since the domain of a network is invariant under reduction, the reduction congruence induced by location observables (denoted by $\simeq_{\mathrm{d}}$) can be decided by a simple syntactic inspection, and is much coarser than the other reduction congruences we are going to consider.

Tree observables. In the setting of dynamic Web data, a natural criterion to decide when two networks are equivalent is to compare the structure of the data tree at each location. We may think of processes as working in the background and not being directly observable. However, an observable which takes into account the exact structure of trees would be overly restrictive for our purposes, because scripts (and queries) can be semantically equivalent without being syntactically (or structurally) equivalent. For this reason we introduce in Figure 4.1 an equivalence relation $\doteq$ on trees which abstracts away from scripts and queries. The reduction congruence induced by the resulting observables will be our reference equivalence.

Definition 4.2.4 (Tree Observables) *Let a* tree observable $l{\cdot}T$ *denote the set*

$$l{\cdot}T \stackrel{\text{def}}{=} \left\{N \;:\; \exists C[-], T', P.\; N \equiv C[l\,[\,T' \parallel P\,]],\; T \doteq T'\right\}$$

The tree observables *are the set* $\mathcal{O}_{\mathrm{t}} \stackrel{\text{def}}{=} \{l{\cdot}T \;:\; l \in \mathcal{L},\, T \in \mathcal{T}\}$.

Figure 4.1: Shape equivalence

$$T_1 \mathrel{\hat{=}} T_1',\, T_2 \mathrel{\hat{=}} T_2' \implies \mathtt{a}[\,T_1\,]_{\mathsf{I}}T_2 \mathrel{\hat{=}} \mathtt{a}[\,T_1'\,]_{\mathsf{I}}T_2' \qquad \text{(Shape Tree)}$$

$$T \mathrel{\hat{=}} T' \implies \mathtt{a}[\,p@l\,]_{\mathsf{I}}T \mathrel{\hat{=}} \mathtt{a}[\,p'@l\,]_{\mathsf{I}}T' \qquad \text{(Shape Pointer)}$$

$$T \mathrel{\hat{=}} T' \implies \mathtt{a}[\,\langle A\rangle\,]_{\mathsf{I}}T \mathrel{\hat{=}} \mathtt{a}[\,\langle A'\rangle\,]_{\mathsf{I}}T' \qquad \text{(Shape Script)}$$

Shape equivalence $\mathrel{\hat{=}}$ is a subset of $\mathcal{T} \times \mathcal{T}$. It is the least equivalence relation satisfying the rules given above.

The induced reduction congruence $\simeq_{\mathrm{t}}$ depends on the query language. For example, if the queries are read-only (processes cannot modify trees), $\simeq_{\mathrm{t}}$ can be decided statically, like $\simeq_{\mathrm{d}}$. Note that, by point 2 of Lemma 4.1.4, it is always the case that $\simeq_{\mathrm{t}} \subseteq \simeq_{\mathrm{d}}$, even for a choice of the equivalence relation $\mathrel{\hat{=}}$ diffrent from the one given in Figure 4.1.

If the query language is at least as expressive as Sam then $\simeq_{\mathrm{t}}$ becomes non-trivial. In particular, thanks to context closure, $\simeq_{\mathrm{t}}$ has the power to discriminate scripts up-to semantic equivalence. For example

$$l\,[\,\mathtt{a}[\,\langle\overline{c}\rangle\,]\parallel \mathbf{0}\,] \not\simeq_{\mathrm{t}} l\,[\,\mathtt{a}[\,\langle\overline{b}\rangle\,]\parallel \mathbf{0}\,]$$

because the context $m\,[\,\varnothing \parallel \mathsf{go}\, l.(\mathsf{run}_{\mathtt{a}}\langle\rangle \mid c.\mathsf{req}_{\mathsf{cut}(x)}\langle d\rangle)\,] \mid -$ can tell the two networks apart. The context sends to l a process which executes the script and then waits on channel c for a message to trigger the removal of the current tree at l. This message is provided by the script on the left hand side, but not by the one on the right hand side.

Process observables. Our processes are inspired by the asynchronous π-calculus. In that language, the natural observables are the output barbs, which distinguish processes based on their ability to perform an output action on a free channel. Asynchronous π-calculus processes can only interact via communication, hence the equivalence based on output barbs characterizes the ability of a context to tell apart two processes. Matters are more complex in X$d\pi$ where processes can also interact with the store: we study both an equivalence based on output barbs and one based on requests to the store.

Definition 4.2.5 (Channel Observables) *Let a* channel observable $l{\cdot}a$ *denote the set*

$$l{\cdot}a \stackrel{\text{def}}{=} \left\{N \;:\; \exists C[-], T, \widetilde{v}, P.\; N \equiv C[l\,[\,T \parallel \overline{a}\langle\widetilde{v}\rangle \mid P\,]],\; a \in \mathit{fn}(N)\right\}$$

The channel observables *are the set* $\mathcal{O}_{\mathrm{c}} \stackrel{\text{def}}{=} \{l{\cdot}a \;:\; l \in \mathcal{L},\, a \in \mathcal{C}_{\mathrm{p}} \cup \mathcal{C}_{\mathrm{s}}\}$.

Definition 4.2.6 (Request Observables) *Let a* request observable $l{\cdot}p$ *denote the set*

$$l{\cdot}p \stackrel{\text{def}}{=} \left\{N \;:\; \exists C[-], T, c, P.\; N \equiv C[l\,[\,T \parallel \mathsf{req}_p\langle c\rangle \mid P\,]]\right\}$$

The request observables *are the set* $\mathcal{O}_{\mathrm{r}} \stackrel{\text{def}}{=} \{l{\cdot}p \;:\; l \in \mathcal{L},\, p \in \mathcal{Q}\}$.

Both $\simeq_{\mathrm{c}}$ and $\simeq_{\mathrm{r}}$ depend on the query language chosen. For example, if we considered an instance of X$d\pi$ where the store is write only (query evaluation can modify the store but always returns the empty result), then the two networks below would be equivalent:

$$l\,[\,\mathsf{a}[\,T\,]\parallel \mathsf{req}_p\langle c\rangle\,] \simeq_{\mathrm{c}} l\,[\,\varnothing \parallel \overline{c}\langle\varnothing\rangle\,].$$

This is not very intuitive, as the stores can be arbitrarily different. If the query language is at least as powerful as Sam, then examples such as the one above are ruled out and, quite remarkably, $\simeq_{\mathrm{c}}$ and $\simeq_{\mathrm{t}}$ coincide.

The situation is different when we look at $\simeq_{\mathrm{r}}$. Consider the inequality given below

$$l\,[\,T \parallel (\nu\, c)\mathsf{req}_{\mathsf{copy}(x)}\langle c\rangle\,] \not\simeq_{\mathrm{r}} l\,[\,T \parallel \mathbf{0}\,]$$

where $\mathsf{copy}(x)$ does not modify the store. Both $\simeq_{\mathrm{t}}$ and $\simeq_{\mathrm{c}}$ regard these networks as equivalent, because after reduction the store remains unchanged and the results of the request are returned on a restricted channel which cannot be used by any other process. This suggests that $\simeq_{\mathrm{r}}$ may be more discriminating than $\simeq_{\mathrm{t}}$ and $\simeq_{\mathrm{c}}$, as long as the query language contains a query similar to $\mathsf{copy}(x)$. In the next section, we formalize the relationship between these equivalence relations.

4.2.2 Comparing the equivalences

X$d\pi$ processes can use pattern matching to discriminate between different tree values. In order to compare formally the expressivity of $\simeq_{\mathrm{t}}$, $\simeq_{\mathrm{c}}$ and $\simeq_{\mathrm{r}}$, the first step is to understand the relation between shape equivalence $\doteq$ and pattern matching. In order to do so, we introduce the notion of *defining pattern* for a given tree, which intuitively represents its shape.

Definition 4.2.7 (Defining Pattern) *Given a tree T, the pattern π is a* defining pattern *for T if the judgment $\mathcal{D}(T) = \pi$ can be derived using the following rules*

$$\begin{aligned}
\mathcal{D}(\varnothing) &= \varnothing \\
\mathcal{D}(\mathtt{a}[\,T_1\,]_l T_2) &= \mathtt{a}[\,\mathcal{D}(T_1)\,]_l \mathcal{D}(T_2) \text{ where } \mathit{fv}(\mathcal{D}(T_1)) \cap \mathit{fv}(\mathcal{D}(T_2)) = \emptyset \\
\mathcal{D}(\mathtt{a}[\,p@l\,]_l T) &= \mathtt{a}[\,x@l\,]_l \mathcal{D}(T) \text{ where } x \notin \mathit{fv}(\mathcal{D}(T)) = \emptyset \\
\mathcal{D}(\mathtt{a}[\,\langle A\rangle\,]_l T) &= \mathtt{a}[\,\langle x\rangle\,]_l \mathcal{D}(T) \text{ where } x \notin \mathit{fv}(\mathcal{D}(T)) = \emptyset.
\end{aligned}$$

It turns out that two trees are shape-equivalent if and only if they pattern match against the same defining pattern.

Proposition 4.2.8 (Shape Equivalence) *For all trees T and S,*

1. *if $T \doteq S$, then for any defining pattern $\pi = \mathcal{D}(T)$ there are two substitutions σ, ρ such that $T = \pi\sigma$ and $S = \pi\rho$, where $\mathit{dom}(\sigma) = \mathit{dom}(\rho) = \mathit{fv}(\pi)$;*
2. *if there are a defining pattern $\pi = \mathcal{D}(T)$ and two substitutions σ, ρ such that $T = \pi\sigma$ and $S = \pi\rho$, where $\mathit{dom}(\sigma) = \mathit{dom}(\rho) = \mathit{fv}(\pi)$, then $T \doteq S$.*

Proof.

1. The proof is by induction on the structure of T.

   - ($T = \varnothing$) If $T \doteq S$, it must be the case that $S = \varnothing$. By definition, $\varnothing$ is the only defining pattern for T. Any σ, ρ are such that $T = \pi\sigma$ and $S = \pi\rho$.
   - ($T = \mathtt{a}[\,T_1\,]_l T_2$) If $T \doteq S$, it must be the case that rule (Shape Tree) was applied, hence $S = \mathtt{a}[\,S_1\,]_l S_2$, $T_1 \doteq S_1$ and $T_2 \doteq S_2$. By definition, any $\pi = \mathcal{D}(T)$ has the from $\mathtt{a}[\,\pi_1\,]_l \pi_2$ where $\pi_1 = \mathcal{D}(T_1)$, $\pi_2 = \mathcal{D}(T_2)$ and $\mathit{fv}(\pi_1) \cap \mathit{fv}(\pi_2) = \emptyset$. By inductive hypothesis, since $T_1 \doteq S_1$, we have that for all $\pi = \mathcal{D}(T_1)$, hence in particular for π_1, there are σ_1, ρ_1 such that $T_1 = \pi_1\sigma_1$ and $S_1 = \pi_1\rho_1$. The same reasoning applies to T_2, S_2. By construction, the substitutions $\sigma = \sigma_1 \cup \sigma_2$ and $\rho = \rho_1 \cup \rho_2$ are such that $T = \pi\sigma$ and $S = \pi\rho$.
   - ($T = \mathtt{a}[\,p@l\,]_l T_1$) If $T \doteq S$, it must be the case that rule (Shape Pointer) was applied, hence $S = \mathtt{a}[\,q@l\,]_l S_1$ and $T_1 \doteq S_1$. By definition, any $\pi = \mathcal{D}(T)$ has the from $\mathtt{a}[\,x@l\,]_l \pi_1$ where $\pi_1 = \mathcal{D}(T_1)$ and $x \notin \mathit{fv}(\pi_1)$. By inductive hypothesis, since $T_1 \doteq S_1$, we have that for any defining pattern for T_1, and in particular for π_1, there are σ_1, ρ_1 such that $T_1 = \pi_1\sigma_1$ and $S_1 = \pi_1\rho_1$. By construction, the substitutions $\sigma = \{^p/_x\} \cup \sigma_1$ and $\rho = \{^q/_x\} \cup \rho_1$ are such that $T = \pi\sigma$ and $S = \pi\rho$.

- $(T = \mathtt{a}[\,\langle A \rangle\,]_{\mid} T_1)$ Similar to the previous case.

2. The proof is by induction on the structure of T, similar to the one given above. We show only the most interesting case. Suppose $T = \mathtt{a}[\,T_1\,]_{\mid} T_2$. We assume that there is a $\pi = \mathcal{D}(T)$ such that, for all S, there are σ, ρ with $T = \pi\sigma$ and $S = \pi\rho$. By construction, it must be the case that $\pi = \mathtt{a}[\,\pi_1\,]_{\mid}\pi_2$ where $\mathit{fv}(\pi_1) \cap \mathit{fv}(\pi_2) = \emptyset$, $\pi_1 = \mathcal{D}(T_1)$ and $\pi_2 = \mathcal{D}(T_2)$. Since $T = \pi\sigma$, it must be the case that $T_i = \pi_i\sigma_i$ and $\sigma = \sigma_1 \cup \sigma_2$. Since $S = \pi\rho$, it must be the case that $S = \mathtt{a}[\,S_1\,]_{\mid} S_2$ where $S_i = \pi_i\rho_i$ and $\rho = \rho_1 \cup \rho_2$. By inductive hypothesis, since $\pi_i = \mathcal{D}(T_i)$, $T_i = \pi_i\sigma_i$ and $S_i = \pi_i\rho_i$, then $T_i \doteq S_i$. By (Shape Tree), $T \doteq S$.

□

We want to compare the expressive power of the different network equivalences, but we have noted above that both $\simeq_{\mathrm{c}}$ and $\simeq_{\mathrm{r}}$ depend on the query language chosen. Rather than restricting our results to a particular choice of query language, we define below some generic properties on which to base the comparison.

Definition 4.2.9 *A query language $(\mathcal{Q}, \mathit{fv}, \mathfrak{E})$, respecting Definition 2.2.1:*

- *admits* trivial updates *if there are queries which leave some input tree unchanged:*
$$\exists p, T.\ \mathfrak{E}(p, T) = (T, L).$$
- *is* shape-preserving *if it never changes the shape of the input tree:*
$$\forall p, T.\ \mathfrak{E}(p, T) = (T', L) \Longrightarrow T \doteq T'$$
- *is* shape-aware *if for any tree T there is a query p which characterizes its shape, in the sense that the query-result of p on an arbitrary tree T' is shape equivalent to the query-result of p on T if and only if T' is shape equivalent to T:*
$$\forall T.\exists p.\forall T'.\, T \doteq T' \iff \left(\left.\begin{array}{l} \mathfrak{E}(p, T) = (T_1, U_1 \mid \ldots \mid U_n \mid \varnothing), \\ T_1' = \mathtt{r}[\,U_1\,]_{\mid} \ldots {}_{\mid}\mathtt{r}[\,U_n\,]_{\mid}\varnothing, \\ \mathfrak{E}(p, T') = (T_2, V_1 \mid \ldots \mid V_m \mid \varnothing), \\ T_2' = \mathtt{r}[\,V_1\,]_{\mid} \ldots {}_{\mid}\mathtt{r}[\,V_m\,]_{\mid}\varnothing \end{array}\right\} \text{ and } T_1' \doteq T_2' \right)$$

Pure query languages as intended in databases are shape-preserving, as they do not modify the input tables, but just return a new table of results. Query languages with update operations instead can modify the input data. Shape-awareness basically requires some agreement between the query language and the data model. If a query language is not shape-aware, then there are documents which have a different structure but which cannot be distinguished by looking at the results of a query. This may indicate that the data model, at least for the sake of querying, contains more information than strictly necessary.

In general, we expect a useful query and update language to be shape-aware but not shape-preserving, and most likely to admit trivial updates.

Proposition 4.2.10 Sam *(i) is not shape-preserving, (ii) is shape-aware and (iii) admits trivial updates.*

Proof. For (i) we have $\mathfrak{E}(\mathsf{cut}(x), \mathtt{a}\,|\varnothing) = (\varnothing, (\mathtt{a}\,|\varnothing)|\varnothing)$ and $\mathtt{a}\,|\varnothing \not\simeq \varnothing$. For (ii), consider arbitrary T, T'. The query $\mathsf{copy}(x)$ gives respectively the results $(T)|\varnothing$ and $(T')|\varnothing$. Suppose $T \simeq T'$. By (Shape Tree), $\mathtt{r}[\,T\,]|\varnothing \simeq \mathtt{r}[\,T'\,]|\varnothing$. Suppose $T \not\simeq T'$ and $\mathtt{r}[\,T\,]|\varnothing \simeq \mathtt{r}[\,T'\,]|\varnothing$. Then we must have applied (Shape Tree) with premise $T \simeq T'$, reaching a contradiction. For (iii) we have that $\mathfrak{E}(\mathsf{copy}(x), T) = (T, T|\varnothing)$.

□

We can now present the main result of this section, which establishes the relative expressivity of the reduction congruences induced by tree, channel and request observables.

Theorem 4.2.11 *(Hierarchy) Depending on the expressive power of the query language, we obtain different inclusions:*

1. *for any query language,* $\simeq_{\mathrm{r}} \subseteq \simeq_{\mathrm{c}}$ *;*
2. *if the query language admits trivial updates then* $\simeq_{\mathrm{r}} \subsetneq \simeq_{\mathrm{c}}$ *;*
3. *if the query language is not shape-preserving then* $\simeq_{\mathrm{t}} \subseteq \simeq_{\mathrm{c}}$ *;*
4. *if the query language is shape-aware, then* $\simeq_{\mathrm{c}} \subseteq \simeq_{\mathrm{t}}$ *.*

Proof.

1. Suppose $N \simeq_{\mathrm{r}} M$. By Lemma 4.1.3, it is enough to show that if $N \downarrow_{l\cdot a}$ we can define a context $C[-]$ such that $C[N] \Downarrow_{m\cdot p}$ and $C[M] \Downarrow_{m\cdot p} \Longrightarrow M \Downarrow_{l\cdot a}$. By definition of $\simeq_{\mathrm{c}}$, $N \equiv (\nu\, \widetilde{c})(N_1 \,|\, l\,[\,T \parallel \overline{a}\langle v_1, \ldots, v_n\rangle \,|\, P\,])$ where $a \notin \{\widetilde{c}\}$. For some fresh m, let

   $$C[-] = m\,[\,\varnothing \parallel \mathsf{go}\, l.a(x_1, \ldots, x_n).\mathsf{go}\, m.\mathsf{req}_p\langle c\rangle\,] \,|\, -$$

   By definition of $\longrightarrow$,

   $$C[N] \longrightarrow\longrightarrow\longrightarrow m\,[\,\varnothing \parallel \mathsf{req}_p\langle c\rangle\,] \,|\, (\nu\, \widetilde{c})(N_1 \,|\, l\,[\,T \parallel P\,])$$

   By definition of $\simeq_{\mathrm{r}}$, $C[N] \Downarrow_{m\cdot p}$.

   Suppose now $C[M] \Downarrow_{m\cdot p}$. Since m is fresh, the only possibility to get $C[M] \overset{*}{\longrightarrow} M' \downarrow_{m\cdot p}$ is by consuming the prefix $\mathsf{go}\, l.a(x_1, \ldots, x_n).\mathsf{go}\, m$ reducing $C[M]$ to $M' \equiv m\,[\,\varnothing \parallel \mathsf{req}_p\langle c\rangle\,] \,|\, M''$. In order to consume the prefix $\mathsf{go}\, l.a(x_1, \ldots, x_n)$, we must have

   $$M \overset{*}{\longrightarrow} M''' \equiv (\nu\, \widetilde{b})(M_1 \,|\, l\,[\,T' \parallel \overline{a}\langle u_1, \ldots, u_n\rangle \,|\, Q_1\,])$$

   where $a \notin \{\widetilde{b}\}$. By definition of $\simeq_{\mathrm{c}}$, $M''' \downarrow_{l\cdot a}$. By definition of $\Downarrow$, $M \Downarrow_{l\cdot a}$.

2. Follows from point 1 and the counterexample $l\,[\,T \parallel (\nu\, c)\mathsf{req}_p\langle c\rangle\,] \not\simeq_{\mathrm{r}} l\,[\,T \parallel \mathbf{0}\,]$ where p, T are such that $\mathfrak{E}(p, T) = (T, L)$ for some list of results L.

3. Suppose $N \simeq_{\mathrm{t}} M$. Since the query language is not shape-preserving, there exist p, T such that $\mathfrak{E}(p,T) = (T', L)$ and $T \not\doteq T'$. By Lemma 4.1.3, we need to show that if $N \downarrow_{l\cdot a}$ we can define a context $C[-]$ such that $C[N] \Downarrow_{m\cdot T'}$ and $C[M] \Downarrow_{m\cdot T'} \Longrightarrow M \Downarrow_{l\cdot a}$. By definition of $\simeq_{\mathrm{c}}$,

$$N \equiv (\nu\, \widetilde{c})(N_1 \mid l\,[\,T \parallel \overline{a}\langle v_1, \ldots, v_n\rangle \mid P\,])$$

where $a \notin \{\widetilde{c}\}$. For some fresh m, let

$$C[-] = m\,[\,T \parallel \mathsf{go}\, l.a(x_1, \ldots, x_n).\mathsf{go}\, m.\mathsf{req}_p\langle c\rangle\,] \mid -$$

By definition of $\longrightarrow$,

$$C[N] \longrightarrow\longrightarrow\longrightarrow\longrightarrow m\,[\,T' \parallel \mathbf{0}\,] \mid (\nu\, \widetilde{c})(N_1 \mid l\,[\,T \parallel P\,])$$

By definition of $\simeq_{\mathrm{r}}$, $C[N] \Downarrow_{m\cdot T'}$.

Suppose now $C[M] \Downarrow_{m\cdot T'}$. Since m is fresh, the only possibility to get $C[M] \longrightarrow^{*} M' \downarrow_{m\cdot T'}$ is by consuming the process $\mathsf{go}\, l.a(x_1, \ldots, x_n).\mathsf{go}\, m.\mathsf{req}_p\langle c\rangle$ reducing $C[M]$ to $M' \equiv m\,[\,T' \parallel \mathbf{0}\,] \mid M''$. In order to consume the prefix $\mathsf{go}\, l.a(x_1, \ldots, x_n)$, we must have

$$M \longrightarrow^{*} M''' \equiv (\nu\, \widetilde{b})(M_1 \mid l\,[\,T_1 \parallel \overline{a}\langle u_1, \ldots, u_n\rangle \mid Q_1\,])$$

where $a \notin \{\widetilde{b}\}$. By definition of $\simeq_{\mathrm{c}}$, $M''' \downarrow_{l\cdot a}$. By definition of $\Downarrow$, $M \Downarrow_{l\cdot a}$.

4. Suppose $N \simeq_{\mathrm{c}} M$. By Lemma 4.1.3, we need to show that if $N \downarrow_{l\cdot T}$ we can find a context $C[-]$ and a barb $\Downarrow_{m\cdot b}$ such that $C[N] \Downarrow_{m\cdot b}$ and $C[M] \Downarrow_{m\cdot b} \Longrightarrow M \Downarrow_{l\cdot T}$. By definition of $\downarrow_{l\cdot T}$, $N \equiv (\nu\, \widetilde{c})(N_1 \mid l\,[\,T' \parallel P\,])$ where $T' \doteq T$. Since the query language is shape-aware, there is p such that, for all T'', $T' \doteq T''$ if and only if

$$\mathfrak{E}(p, T') = (T_1, U_1 \mathbin{\mathtt{|}} \ldots \mathbin{\mathtt{|}} U_n \mathbin{\mathtt{|}} \varnothing), \quad T_1' = \mathtt{r}[\,U_1\,] \mathbin{\mathtt{|}} \ldots \mathbin{\mathtt{|}} \mathtt{r}[\,U_n\,] \mathbin{\mathtt{|}} \varnothing$$

$$\mathfrak{E}(p, T'') = (T_2, V_1 \mathbin{\mathtt{|}} \ldots \mathbin{\mathtt{|}} V_m \mathbin{\mathtt{|}} \varnothing), \quad T_2' = \mathtt{r}[\,V_1\,] \mathbin{\mathtt{|}} \ldots \mathbin{\mathtt{|}} \mathtt{r}[\,V_m\,] \mathbin{\mathtt{|}} \varnothing$$

and $T_1' \doteq T_2'$. Let p be as described above, let $\pi = \mathcal{D}(T_1')$ and, for some fresh m, let

$$C[-] = m\,[\,\varnothing \parallel \mathsf{go}\, l.(\nu\, a)(\mathsf{req}_p\langle a\rangle \mid a(\pi).\mathsf{go}\, m.\overline{b})\,] \mid -$$

Since π is a defining pattern for T_1', we know that there is a σ such that $T_1' = \pi\sigma$. Hence, by definition of $\longrightarrow$,

$$C[N] \longrightarrow^{*\equiv} m\,[\,\varnothing \parallel \overline{b}\,] \mid (\nu\, \widetilde{c})(N_1 \mid l\,[\,T_1 \parallel P\,])$$

By definition of $\simeq_{\mathrm{c}}$, $C[N] \Downarrow_{m\cdot b}$.

Suppose now $C[M] \Downarrow_{m\cdot b}$. We need to show that $M \Downarrow_{l\cdot T}$. Since m is fresh, the only possibility to obtain $C[M] \longrightarrow^{*} M' \downarrow_{m\cdot b}$ is by consuming the prefix $\mathsf{go}\, l.(\nu\, a)(\mathsf{req}_p\langle a\rangle \mid a(\pi).\mathsf{go}\, m.-)$ reducing $C[M]$ to

$$M' \equiv m\,[\,\varnothing \parallel \overline{b}\,] \mid (\nu\, \widetilde{c'})(M_1 \mid l\,[\,T_3 \parallel Q_1\,])$$

In order to reach this network, we must have that

$$M \longrightarrow^{*} M'' \equiv (\nu\, \widetilde{c'})(M_1 \mid l\,[\,T''' \parallel Q_1\,])$$

By the shape awareness condition reported above, $\mathsf{req}_p\langle a\rangle$ returns on channel a a result $T_3' \doteq T_1'$ if and only if $T''' \doteq T'$. Moreover, by Proposition 4.2.8, $T_3' = \pi\rho$ for some ρ (and pattern matching can succeed) if and only if $T_1' \doteq T_3'$. Hence, we have

$$C[M] \longrightarrow^{*} C[M''] \longrightarrow$$

$$m\,[\,\varnothing \parallel \mathbf{0}\,] \mid (\nu\, \widetilde{c'})(M_1 \mid l\,[\,T''' \parallel (\nu\, a)(\mathsf{req}_p\langle a\rangle \mid a(\pi).\mathsf{go}\, m.\bar{b}) \mid Q_1\,]) \longrightarrow$$

$$m\,[\,\varnothing \parallel \mathbf{0}\,] \mid (\nu\, \widetilde{c'})(M_1 \mid l\,[\,T_3 \parallel (\nu\, a)(\overline{a}\langle T_3'\rangle \mid a(\pi).\mathsf{go}\, m.\bar{b}) \mid Q_1\,]) \longrightarrow\longrightarrow M'$$

By transitivity of $\doteq$, $T \doteq T'''$. By definition of $\simeq_{\mathrm{c}}$, $M'' \downarrow_{l\cdot T}$. By definition of $\Downarrow$, $M \Downarrow_{l\cdot T}$.

□

Once again, we instantiate the general result to Sam.

Corollary 4.2.12 *For* Sam*, we have* $\simeq_{\mathrm{r}} \subsetneq \simeq_{\mathrm{c}} = \simeq_{\mathrm{t}}$.

Proof. Follows from Proposition 4.2.10 and Theorem 4.2.11. □

Both $\simeq_{\mathrm{t}}$, $\simeq_{\mathrm{c}}$ and $\simeq_{\mathrm{r}}$ are based on very large sets of observables. We conclude this section with two remarks on some equivalent definitions for tree and channel observables involving less universal quantifications.

Remark 4.2.13 (Less Observables) *We know from point 1 of Lemma 4.1.4 that starting from a given induced congruence, if we consider more specific properties (smaller sets of observables), the resulting induced congruence distinguishes more terms than the original one. For example, we can look only at located empty trees: if* $\mathcal{O}_{\varnothing} \stackrel{\text{def}}{=} \{l\cdot\varnothing \;:\; l \in \mathcal{L}\}$ *then* $\mathcal{O}_{\varnothing} \subseteq \mathcal{O}_{\mathrm{t}}$*, hence* $\simeq_{\mathrm{t}} \subseteq \simeq_{\varnothing}$*. Moreover, if we consider a query language (such as* Sam*) which is shape aware, and contains a query* p *such that, for some* $T \neq \varnothing$*,* $\mathfrak{E}(p, T) = (\varnothing, L)$*, then we can show that* $\simeq_{\varnothing} \subseteq \simeq_{\mathrm{t}}$ *(and hence* $\simeq_{\mathrm{t}} = \simeq_{\varnothing}$*) by adapting the argument of Theorem 4.3.15 for showing that* $\simeq_{\mathrm{c}} \subseteq \simeq_{\mathrm{t}}$*. The case for observing a single channel name is even simpler. Given an arbitrary private channel name* a*, consider* $\mathcal{O}_a \stackrel{\text{def}}{=} \{l\cdot a \;:\; l \in \mathcal{L}\}$*. It is always the case that* $\simeq_a = \simeq_{\mathrm{c}}$*, because the argument for* $\simeq_{\mathrm{t}} \subseteq \simeq_{\mathrm{c}}$ *can be adapted independently from the query language chosen.*

Remark 4.2.14 (Existential Observables) *Let us consider the existential closure of tree observables. If* $\mathcal{O}_{\exists\mathrm{T}} \stackrel{\text{def}}{=} \left\{\bigcup_{T\in\mathcal{T}} l\cdot T \;:\; l \in \mathcal{L}\right\}$ *then we know from point 2 of Lemma 4.1.4 that* $\simeq_{\mathrm{t}} \subseteq \simeq_{\exists\mathrm{T}}$*. It is easy to see that independently from the query language,* $\simeq_{\exists\mathrm{T}} = \simeq_{\mathrm{d}}$*, hence typically* $\simeq_{\mathrm{t}} \subsetneq \simeq_{\exists\mathrm{T}}$*. Due to the restriction operator, the situation is different for channel observables. If* $\mathcal{O}_{\exists\mathrm{c}} \stackrel{\text{def}}{=} \left\{\bigcup_{a\in\mathcal{C}_{\mathrm{p}}} l\cdot a \;:\; l \in \mathcal{L}\right\}$ *then from Lemma 4.1.4,* $\simeq_{\mathrm{c}} \subseteq \simeq_{\exists\mathrm{c}}$*, and (following a similar proof appearing in [27]) we can use restriction to build a context such that from Lemma 4.1.3 we get* $\simeq_{\exists\mathrm{c}} \subseteq \simeq_a$*, hence by Remark 4.2.13,* $\simeq_{\exists\mathrm{c}} = \simeq_{\mathrm{c}}$.

4.3 Core X$d\pi$

Network equivalences dictate when two networks can be considered indistinguishable with respect to the properties represented by a specific set of observables. The next step is to define equivalence relations on processes such that, when we place equivalent processes in the same context, we obtain equivalent networks. Since we use the equivalences for example to optimize the interaction between different locations, we would like to be able to compare several located processes, possibly sharing some private channel names, at the same time. Moreover, we want to make sure that the behaviour of the processes is robust with respect to changes in the data stored in each location and to the behaviour of other processes running in parallel.

It is not straightforward to carry on the kind of reasoning mentioned above directly on X$d\pi$ terms, because locations, processes and data are closely integrated. Instead, we introduce a calculus, called Core X$d\pi$, which serves as an alternative representation of X$d\pi$, where we locate processes explicitly and we separate data from processes. Core X$d\pi$ is tailored to be semantically equivalent to X$d\pi$, and is suitable for expressing a partial specification of a network by means of located processes running in parallel, possibly sharing private names. After defining Core X$d\pi$, we give a translation from X$d\pi$ to Core X$d\pi$ and show that it preserves network equivalences. In the next section, we will define process equivalence on Core X$d\pi$ processes.

4.3.1 Syntax and semantics

To guide the intuition, we anticipate that a X$d\pi$ location $l\,[\,T \parallel P\,]$ corresponds to a Core X$d\pi$ term $(\{l \mapsto T'\}, P')$ where $\{l \mapsto T'\}$ says that at location l there is a tree T' (the Core X$d\pi$ equivalent of T), and P' is like P except that it contains explicit location information.

Trees, queries, values, all the other basic sorts of Core X$d\pi$, and the notions of Definition 2.2.1 (including function $\mathfrak{E}$) are the obvious adaptations of the ones for X$d\pi$, and their sorts are differentiated by subscript $-_{\mathrm{C}}$. For example, Core X$d\pi$ trees have the same structure as X$d\pi$ trees but contain Core X$d\pi$ scripts, and are denoted by $\mathcal{T}_{\mathrm{C}}$.

Located processes. Core X$d\pi$ processes are based on asynchronous ${}^{e}\pi$-processes [17], extended with the X$d\pi$ specific operations of migration, application and request. The formal definition is given in Figure 4.2.

The communication constructs correspond to those found in the ${}^{e}\pi$-calculus: the *output* process $\overline{l{\cdot}b}\langle\tilde{v}\rangle$ denotes a vector of values $\tilde{v}$ waiting to be sent via channel b at location l, the *input* process $l{\cdot}b(\tilde{z}).P$ waits to receive values from an output process via channel b at l, and the *replicated input* is standard. Scripts must always have a variable as the first pattern, and application is defined only when the first parameter passed to the script is a location (representing the place where the script will be running). The `req` and `go` commands are the located version of the same commands for X$d\pi$. Note that in the case of input and replicated input the continuation process must be located at the same location where the input is defined, and in the case of migration the continuation must be located at the destination location[1].

[1] We have represented these conditions as restrictions on the grammar of processes. Alternatively,

Figure 4.2: Syntax: Core X$d\pi$ processes

$\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}$ **::=**		process terms
	$\mathbf{0}$	nil process
	$\boldsymbol{P} \mid \boldsymbol{P}$	composition of processes
	$(\nu\, c)\boldsymbol{P}$	private channel c with scope $\boldsymbol{P}$
	$\overline{\boldsymbol{l}\cdot\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle$	at $\boldsymbol{l}$, output on $\boldsymbol{c}$ of values $\widetilde{\boldsymbol{v}}$
	$\boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{l}\cdot\boldsymbol{P}$	at $\boldsymbol{l}$, input on $\boldsymbol{c}$ of $\widetilde{\pi}$ with continuation $\boldsymbol{P}$ $(\mathit{distinct}(\widetilde{\pi}),\ \mathit{fv}(\widetilde{\pi}) \cap \mathit{fv}(\boldsymbol{l}) = \emptyset)$
	$!\boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{l}\cdot\boldsymbol{P}$	lazy replication of an input process $(\mathit{distinct}(\widetilde{\pi}),\ \mathit{fv}(\widetilde{\pi}) \cap \mathit{fv}(\boldsymbol{l}) = \emptyset)$
	$\boldsymbol{l}\cdot\mathsf{go}\,\boldsymbol{m}.\boldsymbol{m}\cdot\boldsymbol{P}$	at $\boldsymbol{l}$, go to $\boldsymbol{m}$, continue with $\boldsymbol{P}$
	$\boldsymbol{A} \circ \langle \boldsymbol{l}, \widetilde{\boldsymbol{v}}\rangle$	at $\boldsymbol{l}$, run script $\boldsymbol{A}$ with parameters $\widetilde{\boldsymbol{v}}$
	$\boldsymbol{l}\cdot\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle$	at $\boldsymbol{l}$, request query $\boldsymbol{p}$ with return channel $\boldsymbol{c}$

Convention: we write $\boldsymbol{l}\cdot\boldsymbol{P}$ for $\boldsymbol{P}$ when $\mathit{dom}(\boldsymbol{P}) = \{\boldsymbol{l}\}$.

$$P, Q, R \in \mathcal{P}_{\mathrm{C}} \stackrel{\text{def}}{=} \left\{ \boldsymbol{P} \ :\ \mathit{fv}(\boldsymbol{P}) = \emptyset \right\} \qquad \text{(Core X}d\pi\text{ Processes)}$$

$$A \in \mathcal{A}_{\mathrm{C}} \stackrel{\text{def}}{=} \left\{ (x, \widetilde{\pi})\boldsymbol{P} \ :\ \begin{array}{c} \mathit{fn}(\boldsymbol{P}) = \emptyset, \mathit{fv}(\boldsymbol{P}) \subseteq \mathit{fv}(x, \widetilde{\pi}), \\ \mathit{distinct}(x, \widetilde{\pi}),\ \mathit{dom}(\boldsymbol{P}) = \{x\} \end{array} \right\} \qquad \text{(Scripts)}$$

$$\mathcal{K}_{\mathcal{P}} \stackrel{\text{def}}{=} \quad C_{\mathcal{P}}[-] \mathbf{::=} \ - \mid P \mid C_{\mathcal{P}}[-] \mid C_{\mathcal{P}}[-] \mid P \mid (\nu\, c)C_{\mathcal{P}}[-] \qquad \text{(Process Contexts)}$$

The functions $\mathit{fv}, \mathit{fn}, \mathit{dom}$ are defined in Figure A.6 and Figure A.2.
Trees, queries, values, all the other basic sorts and the predicate $\mathit{distinct}$ are defined as for X$d\pi$ (trees contain Core X$d\pi$ scripts).
Notation: $\boldsymbol{l}\cdot\overline{\boldsymbol{m}\cdot\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle \stackrel{\text{def}}{=} \boldsymbol{l}\cdot\mathsf{go}\,\boldsymbol{m}.\overline{\boldsymbol{m}\cdot\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle$.

Networks and stores. A network is represented by a pair (D, P) where the first component (the *store*) is a finite partial function from location names to trees, and the second component is a process. The formal definition is given in Figure 4.3. Interaction between processes and data is always local, as we shall see later from rule (CRed Request) in Figure 4.5. In Figure A.2, we define the function dom giving the domain of both networks, stores and processes. By definition, the domain of a network is the domain of the store, and a network is well-formed if the domain of the process is contained in the domain of the store.

Network contexts are pairs of process and store contexts. For example, if $C_{\mathcal{N}} = (- \uplus B, (\nu\, c)-)$ then $C_{\mathcal{N}}[(D, P)] = (D \uplus B, (\nu\, c)P)$. We omit the subscripts from contexts when no ambiguity can arise. Note that, in order to better reflect the X$d\pi$ network composition, the domain of the process context must be included in the

we could have introduced a well-formedness judgment for the continuation processes.

Figure 4.3: Syntax: Core X$d\pi$ networks

$$D, B \in \mathcal{S} \stackrel{\text{def}}{=} \mathcal{L} \rightharpoonup \mathcal{T}_{\mathrm{C}} \qquad \text{(Stores)}$$

$$N, M \in \mathcal{N}_{\mathrm{C}} \stackrel{\text{def}}{=} \{(D, P) : D \in \mathcal{S},\, P \in \mathcal{P}_{\mathrm{C}},\, \mathit{dom}(P) \subseteq \mathit{dom}(D)\} \qquad \text{(Networks)}$$

$$C_{\mathcal{S}} - ::= - \mid C_{\mathcal{S}}[-] \uplus D \qquad \text{(Store Contexts)}$$

$$\mathcal{K}_{\mathcal{N}} \stackrel{\text{def}}{=} \{(C_{\mathcal{S}}, C_{\mathcal{P}}) : C_{\mathcal{S}} \in \mathcal{K}_{\mathcal{S}},\, C_{\mathcal{P}} \in \mathcal{K}_{\mathcal{P}},\, \mathit{dom}(C_{\mathcal{P}}) \subseteq \mathit{dom}(C_{\mathcal{S}})\} \qquad \text{(Networks Contexts)}$$

$$(C_{\mathcal{S}}[-], C_{\mathcal{P}}[-])[(D, P)] \stackrel{\text{def}}{=} (C_{\mathcal{S}}[D], C_{\mathcal{P}}[P]) \qquad \text{(Context Application)}$$

Notation: $\{l \mapsto T\}(l) \stackrel{\text{def}}{=} T$; $\quad (D \uplus B)(l) \stackrel{\text{def}}{=} \begin{cases} D(l) & \text{if } l \in \mathit{dom}(D) \\ B(l) & \text{if } l \in \mathit{dom}(B) \end{cases}$.

Convention: $D \uplus B$ is defined if and only if $\mathit{dom}(D) \cap \mathit{dom}(B) = \emptyset$.
Function dom is defined in Figure A.2.

Figure 4.4: Semantics: structural congruence for Core X$d\pi$

$$(\nu\, c)\mathbf{0} \equiv \mathbf{0} \qquad \text{(CStruct Res PNil)}$$

$$c \notin \mathit{fn}(\boldsymbol{P}) \Longrightarrow \boldsymbol{P} \mid (\nu\, c)\boldsymbol{Q} \equiv (\nu\, c)(\boldsymbol{P} \mid \boldsymbol{Q}) \qquad \text{(CStruct Res PPar)}$$

$$(\nu\, c)(\nu\, d)\boldsymbol{P} \equiv (\nu\, d)(\nu\, c)\boldsymbol{P} \qquad \text{(CStruct Res PRes)}$$

$$P \equiv Q \Longrightarrow (D, P) \equiv (D, Q) \qquad \text{(CStruct Proc)}$$

Structural congruence $\equiv$ is a subset of $(\boldsymbol{P} \times \boldsymbol{P}) \cup (\mathcal{N}_{\mathrm{C}} \times \mathcal{N}_{\mathrm{C}})$. It is the least equivalence relation satisfying α-conversion and the axioms given above, closed under all the syntactic operators, and such that $(\boldsymbol{P}, \mid, \mathbf{0}, \equiv)$ is a commutative monoid. A complete definition of $\equiv$ can be found in Figure A.5.

domain of the store context.

Reduction semantics. The reduction relation $\longrightarrow$ for Core X$d\pi$ describes process interaction, the interaction between processes and data, and the movement of processes across locations. The formal definition is given in Table 4.5. It relies on a standard notion of structural congruence for processes and networks (analogous to the one for X$d\pi$) defined in Figure 4.4.

Rules (CRed Stay) and (CRed Go) are analogous to the ones for X$d\pi$. Rules (CRed Com) and (CRed Com!) are similar to the standard communication rules for the π-calculus, except that processes only communicate if they are at the same location l, and l is present in the store. Rule (CRed Run) runs a script, passing as the first parameter the name of the location where it is going to run. Rule (CRed Request) provides interaction between processes and data, and is analogous to the one for X$d\pi$.

Network equivalences. The network equivalences for Core X$d\pi$ are the reduction

Figure 4.5: Semantics: reduction relation for Core X$d\pi$

$$(\{l \mapsto T\}, l{\cdot}\mathsf{go}\, l.P \,|\, Q) \;\longrightarrow\; (\{l \mapsto T\}, P \,|\, Q) \qquad \text{(CRed Stay)}$$

$$(\{l \mapsto T\} \uplus \{m \mapsto S\}, l{\cdot}\mathsf{go}\, m.P \,|\, Q) \;\longrightarrow\; (\{l \mapsto T\} \uplus \{m \mapsto S\}, P \,|\, Q) \qquad \text{(CRed Go)}$$

$$(\{l \mapsto T\}, \overline{l{\cdot}c}\langle\widetilde{\pi}\sigma\rangle \,|\, l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \,|\, Q) \;\longrightarrow\; (\{l \mapsto T\}, \boldsymbol{P}\sigma \,|\, Q) \qquad \text{(CRed Com)}$$

$$(\{l \mapsto T\}, \overline{l{\cdot}c}\langle\widetilde{\pi}\sigma\rangle \,|\, !l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \,|\, Q) \;\longrightarrow\; (\{l \mapsto T\}, !l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \,|\, \boldsymbol{P}\sigma \,|\, Q) \qquad \text{(CRed Com!)}$$

$$(\{l \mapsto T\}, (x, \widetilde{\pi})\boldsymbol{P} \circ \langle l, \widetilde{\pi}\sigma\rangle \,|\, Q) \;\longrightarrow\; (\{l \mapsto T\}, \boldsymbol{P}\{{}^{l}/_{x}\}\sigma \,|\, Q) \qquad \text{(CRed Run)}$$

$$\frac{\mathfrak{E}(p, T) = (T', U_1 \mid \ldots \mid U_n \mid \varnothing)}{(\{l \mapsto T\}, l{\cdot}\mathsf{req}_p\langle c\rangle \,|\, Q) \longrightarrow (\{l \mapsto T'\}, \overline{l{\cdot}c}\langle \mathtt{r}[\,U_1\,] \mid \ldots \mid \mathtt{r}[\,U_n\,] \mid \varnothing\rangle \,|\, Q)} \qquad \text{(CRed Request)}$$

$$\frac{N \longrightarrow N'}{C_{\mathcal{N}}[N] \longrightarrow C_{\mathcal{N}}[N']} \qquad \text{(CRed Context)} \qquad \frac{N \equiv M \longrightarrow M' \equiv N'}{N \longrightarrow N'} \qquad \text{(CRed Struct)}$$

Reduction $\longrightarrow$ is a partial relation, subset of $\mathcal{N}_{\mathrm{C}} \times \mathcal{N}_{\mathrm{C}}$.
Convention: in this table c ranges over $\mathcal{C}_{\mathrm{p}} \cup \mathcal{C}_{\mathrm{s}}$.

congruences induced by the observables defined below, based on the reduction system $(\mathcal{N}_{\mathrm{C}}, \longrightarrow)$ and the reduction contexts $\mathcal{K}_{\mathcal{N}}$ of Figure 4.3.

Definition 4.3.1 (Core X$d\pi$ Observables) *Let a* tree observable $l{\cdot}T$ *denote the set*

$$l{\cdot}T \stackrel{\text{def}}{=} \big\{(D, P) \;:\; D(l) = T',\, T \doteq T'\big\}$$

The tree observables *are the set* $\mathcal{O}_{\mathrm{tC}} \stackrel{\text{def}}{=} \{l{\cdot}T \;:\; l \in \mathcal{L},\, T \in \mathcal{T}_{\mathrm{C}}\}$. *Let a* channel observable $l{\cdot}a$ *denote the set*

$$l{\cdot}a \stackrel{\text{def}}{=} \big\{(D, P) \;:\; \exists C[-], \widetilde{v}, Q.\; P \equiv C[\overline{l{\cdot}a}\langle\widetilde{v}\rangle \,|\, Q],\, a \in \mathit{fn}(P)\big\}$$

The channel observables *are the set* $\mathcal{O}_{\mathrm{cC}} \stackrel{\text{def}}{=} \{l{\cdot}a \;:\; l \in \mathcal{L},\, a \in \mathcal{C}_{\mathrm{p}} \cup \mathcal{C}_{\mathrm{s}}\}$. *Let a* request observable $l{\cdot}p$ *denote the set*

$$l{\cdot}p \stackrel{\text{def}}{=} \big\{(D, P) \;:\; \exists C[-], c, Q.\; P \equiv C[\mathsf{req}_p\langle c\rangle \,|\, Q]\big\}$$

The request observables *are the set* $\mathcal{O}_{\mathrm{rC}} \stackrel{\text{def}}{=} \{l{\cdot}p \;:\; l \in \mathcal{L},\, p \in \mathcal{Q}_{\mathrm{C}}\}$.

The notation for Core X$d\pi$ observables is the same as the one given in Section 4.2 for X$d\pi$. It will be clear from the context which observables we are referring to. Similarly, we use $\simeq_{\mathrm{t}}$, $\simeq_{\mathrm{c}}$ and $\simeq_{\mathrm{r}}$ to denote also the reduction congruences for Core X$d\pi$ induced by tree, channel and request observables.

4.3.2 From X$d\pi$ to Core X$d\pi$

We now formally define how to translate X$d\pi$ terms into Core X$d\pi$ terms. First we give encodings for networks, values and processes, which are independent from the query language, then we discuss the encoding of the queries.

Figure 4.6: Encodings from X$d\pi$ to Core X$d\pi$

Network translation:

$(\![\mathbf{0}]\!) = (\emptyset, \mathbf{0})$

$(\![N \mid M]\!) = (D \uplus B, P \mid Q)$ where $(\![N]\!) = (D, P)$ and $(\![M]\!) = (B, Q)$

$(\![(\nu\, c)N]\!) = (D, (\nu\, c)P)$ where $(\![N]\!) = (D, P)$

$(\![l\,[\,T \parallel P\,]]\!) = (\{l \mapsto [\![T]\!]\}, \langle\!|P|\!\rangle_l)$

Value translation:

$[\![\boldsymbol{E}\mathbin{!}\boldsymbol{T}]\!] = [\![\boldsymbol{E}]\!]\mathbin{!}[\![\boldsymbol{T}]\!]$

$[\![\varnothing]\!] = \varnothing$

$[\![\mathsf{a}[\,\boldsymbol{V}\,]]\!] = \mathsf{a}[\,[\![\boldsymbol{V}]\!]\,]$

$[\![\boldsymbol{p}@\boldsymbol{l}]\!] = [\![\boldsymbol{p}]\!]@\boldsymbol{l}$

$[\![\langle \boldsymbol{A} \rangle]\!] = \langle [\![\boldsymbol{A}]\!] \rangle$

$[\![c]\!] = c$, $[\![l]\!] = l$, $[\![p]\!] = \{\!|p|\!\}$

$[\![x]\!] = x$, $[\![(\widetilde{\pi})\boldsymbol{P}]\!] = (x, \widetilde{\pi})\langle\!|\boldsymbol{P}|\!\rangle_x$, $\mathit{distinct}(x, \widetilde{\pi})$

$[\![\boldsymbol{v}', \widetilde{\boldsymbol{v}}]\!] = [\![\boldsymbol{v}']\!], [\![\widetilde{\boldsymbol{v}}]\!]$

Substitutions:

$[\![\{\widetilde{v}/\widetilde{x}\}]\!] = \{[\![\widetilde{v}]\!]/\widetilde{x}\}$

Process translation:

$\langle\!|\mathbf{0}|\!\rangle_{\boldsymbol{l}} = \mathbf{0}$

$\langle\!|\boldsymbol{P} \mid \boldsymbol{Q}|\!\rangle_{\boldsymbol{l}} = \langle\!|\boldsymbol{P}|\!\rangle_{\boldsymbol{l}} \mid \langle\!|\boldsymbol{Q}|\!\rangle_{\boldsymbol{l}}$

$\langle\!|(\nu\, c)\boldsymbol{P}|\!\rangle_{\boldsymbol{l}} = (\nu\, c)\langle\!|\boldsymbol{P}|\!\rangle_{\boldsymbol{l}}$

$\langle\!|\mathsf{go}\ \boldsymbol{m}.\boldsymbol{P}|\!\rangle_{\boldsymbol{l}} = \boldsymbol{l}\cdot\mathsf{go}\,\boldsymbol{m}.\langle\!|\boldsymbol{P}|\!\rangle_{\boldsymbol{m}}$

$\langle\!|\overline{\boldsymbol{a}}\langle\widetilde{\boldsymbol{v}}\rangle|\!\rangle_{\boldsymbol{l}} = \overline{\boldsymbol{l}\cdot\boldsymbol{a}}\langle[\![\widetilde{\boldsymbol{v}}]\!]\rangle$

$\langle\!|\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{P}|\!\rangle_{\boldsymbol{l}} = \boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\langle\!|\boldsymbol{P}|\!\rangle_{\boldsymbol{l}}$

$\langle\!|!\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{P}|\!\rangle_{\boldsymbol{l}} = !\boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\langle\!|\boldsymbol{P}|\!\rangle_{\boldsymbol{l}}$

$\langle\!|\boldsymbol{A} \circ \langle\widetilde{\boldsymbol{v}}\rangle|\!\rangle_{\boldsymbol{l}} = [\![\boldsymbol{A}]\!] \circ \langle \boldsymbol{l}, [\![\widetilde{\boldsymbol{v}}]\!]\rangle$

$\langle\!|\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle|\!\rangle_{\boldsymbol{l}} = \boldsymbol{l}\cdot\mathsf{req}_{[\![\boldsymbol{p}]\!]}\langle\boldsymbol{c}\rangle$

Lists:

$[\![L_1\mathbin{!}\ldots\mathbin{!}L_n]\!] = [\![L_1]\!]\mathbin{!}\ldots\mathbin{!}[\![L_n]\!]$.

Networks, values and processes. The encoding of X$d\pi$ into Core X$d\pi$ is defined in Figure 4.6. The encoding of networks is straightforward, the only point worth noting is that the translation of a process depends on the location where it is found. For example, $(\![l\,[\,\varnothing \parallel \overline{a}\,]]\!) = (\{l \mapsto \varnothing\}, \overline{l\cdot a})$. The encoding of values is a homomorphism, where the only interesting cases are the translation of queries, which invokes a query encoding $\{\!|-|\!\}$ (dependent on the query language at hand), and the translation of scripts, which adds a fresh variable x as the first parameter of the script in order to record the location at which the body of the script is translated.

The encoding of processes is parametric in the location placeholder l where the process is going to run. It is mostly homomorphic, but it should be noted that in order to record where an action takes place, inputs, outputs, migrations and requests are prefixed by the location placeholder l. Note that the translation of application adds l as the first parameter passed to a script (which matches with the variable introduced in the translation of scripts) so that the body of the script becomes located at l. For example,

$$(\![l\,[\,T \parallel (y,z)\overline{y}\langle z\rangle \circ \langle a, b\rangle\,]]\!) \;=\; (\{l \mapsto [\![T]\!]\}, (x,y,z)\overline{x\cdot y}\langle z\rangle \circ \langle l, a, b\rangle).$$

Notice that the reductions of the two terms are in close correspondence

$$l\,[\,T \parallel (y,z)\overline{y}\langle z\rangle \circ \langle a, b\rangle\,] \longrightarrow l\,[\,T \parallel \overline{a}\langle b\rangle\,]$$

$$(\{l \mapsto [\![T]\!]\}, (x,y,z)\overline{x\cdot y}\langle z\rangle \circ \langle l, a, b\rangle) \longrightarrow (\{l \mapsto [\![T]\!]\}, \overline{l\cdot a}\langle b\rangle).$$

Queries. We cannot fix once and for all the encoding $\{\!|-|\!\}$ of queries as we are not committed to a specific query language. We just require any encoding for queries to

be a total function. On the other hand, we can define properties which are useful to guarantee that an encoding preserves at least $\simeq_{\mathrm{r}}$ (and therefore, any less restrictive equivalence).

Definition 4.3.2 (Properties of Query Encoding) *Let $\llbracket - \rrbracket$ be the encoding defined in Figure 4.6. An encoding $\{\!\![-]\!\!\} \in \mathcal{Q} \to \mathcal{Q}_{\mathrm{C}}$ is*

- observation-injective *if whenever $l{\cdot}\{\!\![p]\!\!\} = l{\cdot}\{\!\![q]\!\!\}$ then $l{\cdot}p = l{\cdot}q$;*
- homomorphic *if*
  1. *if $\mathfrak{E}(p, T) = (T', L)$ then $\mathfrak{E}(\{\!\![p]\!\!\}, \llbracket T \rrbracket) = (\llbracket T' \rrbracket, \llbracket L \rrbracket)$;*
  2. *if $\mathfrak{E}(\{\!\![p]\!\!\}, \llbracket T \rrbracket) = (T'', L)$ then there exist T', L' such that $\mathfrak{E}(p, T) = (T', L')$, where $\llbracket T' \rrbracket = T''$ and $\llbracket L' \rrbracket = L$.*

Note that in the definition above, since $l{\cdot}p$ and $l{\cdot}q$ are sets, equality means double inclusion. Moreover, given the definition of request observables, the condition of observable-injectivity coincides with injectivity in the classical sense, being equivalent to $p \neq q \implies \{\!\![p]\!\!\} \neq \{\!\![q]\!\!\}$. Below we define an encoding for Sam which is both observation-injective and homomorphic. The semantics of query evaluation in Core X$d\pi$ is the same as the one defined in Figure 2.10 for X$d\pi$, with the difference that scripts are now Core X$d\pi$ scripts.

Definition 4.3.3 (Query Language Encoding) *The encoding for* Sam *is given by $\{\!\![\widehat{p}(\pi)\boldsymbol{V}]\!\!\} = \widehat{p}(\pi)\llbracket \boldsymbol{V} \rrbracket$.*

The definitions of $\llbracket - \rrbracket$ and $\{\!\![-]\!\!\}$ are mutually recursive, but it is easy to see that the recursion is well-founded because it is guarded by the binding pattern (π). Before showing that the encoding of queries is injective and homomorphic in the sense of Definition 4.3.2, we need to show that the encodings of values and processes respect substitutions.

Lemma 4.3.4 (Encoding Substitutions) *The encodings of values and processes respect substitutions:*

- $\llbracket \boldsymbol{V}\sigma \rrbracket = \llbracket \boldsymbol{V} \rrbracket \llbracket \sigma \rrbracket$;
- $\langle\!\langle \boldsymbol{P}\sigma \rangle\!\rangle_l = \langle\!\langle \boldsymbol{P} \rangle\!\rangle_l \llbracket \sigma \rrbracket$.

Proof. Follows by structural induction on $\boldsymbol{V}$ and $\boldsymbol{P}$. □

Proposition 4.3.5 (Injective and Homomorphic Query Encoding) *The encoding given in Definition 4.3.3 is observation-injective and homomorphic.*

Proof. Observation injectivity follows easily by definition of request observables and by structural induction on the value $\boldsymbol{V}$ contained in a query. We show that $\{\!\![-]\!\!\}$ is homomorphic.

1. Suppose $\mathfrak{E}(\widehat{p}(\pi)\boldsymbol{V}, T) = (T', L)$. We show that $\mathfrak{E}(\widehat{p}(\pi)[\![\boldsymbol{V}]\!], [\![T]\!]) = ([\![T']\!], [\![L]\!])$ by induction on the derivations of $\mathfrak{E}$.

   (Eval Match) Suppose $\mathfrak{E}((\pi)[\![\boldsymbol{V}]\!], [\![T]\!]) = (T'', L')$. By hypothesis $T = \pi\sigma$, $T' = \boldsymbol{V}\sigma$ and $L = \pi\sigma \shortmid \varnothing$ for some σ. By Lemma 4.3.4 and definition of the encoding, $[\![\pi\sigma]\!] = \pi[\![\sigma]\!]$, hence $T'' = [\![\boldsymbol{V}]\!][\![\sigma]\!]$ and $L' = \pi[\![\sigma]\!] \shortmid \varnothing$. By Lemma 4.3.4 and definition of the encoding, $T'' = [\![T']\!]$ and $L' = [\![L]\!]$.

   (Eval Anywhere Tree) Suppose the last instance of rule used is

$$\frac{\begin{array}{c}\mathfrak{E}(\square\widehat{p}(\pi)[\![\boldsymbol{V}]\!], [\![V]\!]) = (V', L) \\ \mathfrak{E}(\square\widehat{p}(\pi)[\![\boldsymbol{V}]\!], [\![T]\!]) = (T', L')\end{array} \quad \mathfrak{E}(\widehat{p}(\pi)[\![\boldsymbol{V}]\!], \mathtt{a}[\, V' \,] \shortmid T') = (T'', L'')}{\mathfrak{E}(\square\widehat{p}(\pi)[\![\boldsymbol{V}]\!], \mathtt{a}[\, [\![V]\!] \,] \shortmid [\![T]\!]) = (T'', L \shortparallel L' \shortparallel L'')}$$

   where we have used $[\![\mathtt{a}[\, V \,] \shortmid T]\!] = \mathtt{a}[\, [\![V]\!] \,] \shortmid [\![T]\!]$. By hypothesis,

$$\frac{\begin{array}{c}\mathfrak{E}(\square\widehat{p}(\pi)\boldsymbol{V}, V) = (V'_1, L_1) \\ \mathfrak{E}(\square\widehat{p}(\pi)\boldsymbol{V}, T) = (T'_1, L'_1)\end{array} \quad \mathfrak{E}(\widehat{p}(\pi)\boldsymbol{V}, \mathtt{a}[\, V'_1 \,] \shortmid T'_1) = (T''_1, L''_1)}{\mathfrak{E}(\square\widehat{p}(\pi)\boldsymbol{V}, \mathtt{a}[\, V \,] \shortmid T) = (T''_1, L_1 \shortparallel L'_1 \shortparallel L''_1)}$$

   By inductive hypothesis, $[\![V'_1]\!] = V'$ and $[\![L_1]\!] = L$, and $[\![T'_1]\!] = T'$ and $[\![L'_1]\!] = L'$. Since $[\![\mathtt{a}[\, V'_1 \,] \shortmid L'_1]\!] = \mathtt{a}[\, V' \,] \shortmid L'$, we can apply the inductive hypothesis also on the third premise, obtaining $[\![T''_1]\!] = T''$, $[\![L''_1]\!] = L''$. By definition of the encoding, we get $[\![L_1 \shortparallel L'_1 \shortparallel L''_1]\!] = L_1 \shortparallel L'_1 \shortparallel L''_1$ and we conclude.

   The other cases are similar.

2. Suppose $\mathfrak{E}(\{[p]\}, [\![T]\!]) = (T'', L)$. By induction on the derivations of $\mathfrak{E}$, reasoning like in the previous case, we get $\mathfrak{E}(p, T) = (T', L')$ where $[\![T']\!] = T''$ and $[\![L']\!] = L$.

□

4.3.3 Properties of the encoding

We now proceed to show that the encoding preserves the reduction congruences induced by a large class of observables. Structural congruence and shape equivalence do not depend on the query language, and are always preserved by $(\![-]\!)$.

Lemma 4.3.6 (Structural Congruence Preservation) *For any $N, M \in \mathcal{N}$, $N \equiv M$ if and only if $(\![N]\!) \equiv (\![M]\!)$.*

Proof. By a simple induction on the derivations of $N \equiv M$ and $(\![N]\!) \equiv (\![M]\!)$. □

Lemma 4.3.7 (Shape Equivalence Preservation) *For any Xdπ tree T, $T \eqsim [\![T]\!]$. For any Core Xdπ tree T', there is an Xdπ tree T such that $[\![T]\!] \eqsim T'$.*

Proof. By a simple induction on the structure of T and T'. □

We break down the problem of comparing reduction congruences for X$d\pi$ and Core X$d\pi$ into comparing the contexts, the reduction steps and the observations.

Contexts. First of all, we note that the encodings of trees, processes and networks are surjective: each Core X$d\pi$ term is at least structurally equivalent to the encoding of an X$d\pi$ term.

Lemma 4.3.8 (Surjective Encodings) *The encodings defined in Figure 4.6 are surjective:*

1. *for any Core X$d\pi$ tree T there is an X$d\pi$ tree T' such that $[\![T']\!] = T$;*
2. *for any Core X$d\pi$ process P with domain l there is an X$d\pi$ process P' such that $(\!|P'|\!)_l = P$;*
3. *for any Core X$d\pi$ network (D, P) there is an X$d\pi$ network N such that $(D, P) \equiv (\!|N|\!)$.*

Proof. The first two points follow by a simple structural induction and by definition of the encodings. The proof for (3) is by induction on the structure of D. If $D = \emptyset$ then since $\mathit{dom}(P) \subseteq \mathit{dom}(D)$ we have $P = (\nu\, \widetilde{c})\mathbf{0}$ and $N = (\nu\, \widetilde{c})\mathbf{0}$, which gives $(D, P) \equiv (\!|N|\!)$. Suppose now $D = D' \uplus \{l \mapsto T\}$. First note that we can always use structural congruence to rearrange (D, P) in the form $(D, (\nu\, \widetilde{c})(l_1{\cdot}P_1 \mid \ldots l_n{\cdot}P_n \mid \mathbf{0}))$, where the l_i are distinct and $\{l_1, \ldots, l_n\} \subseteq \mathit{dom}(D)$. Suppose $l \notin \{l_1, \ldots, l_n\}$. By point (1) of this lemma, there is T' such that $[\![T']\!] = T$. Let $N_l = l\,[\,T' \,\|\, \mathbf{0}\,]$. Since $\{l_1, \ldots, l_n\} \subseteq \mathit{dom}(D)$, by inductive hypothesis there is N' such that $(D', (\nu\, \widetilde{c})(l_1{\cdot}P_1 \mid \ldots l_n{\cdot}P_n \mid \mathbf{0})) \equiv (\!|N'|\!)$. By definition of the encoding, $(D, P) \equiv (\!|N_l \mid N'|\!)$. Suppose instead that there is an i such that $l = l_i$. Since $\{l_1, \ldots, l_n\} \setminus \{l\} \subseteq \mathit{dom}(D')$, by inductive hypothesis there is N' such that $(D', \prod_{j \neq i} l_j{\cdot}P_j \mid \mathbf{0}) \equiv (\!|N'|\!)$. By points (1) and (2) of this lemma, there exist T' and P' such that $[\![T']\!] = T$ and $(\!|P'|\!)_l = P_i$. By definition of the encoding, $(\!|(\nu\, \widetilde{c})(l\,[\,T' \,\|\, P'\,] \mid N')|\!) \equiv (D, P)$. □

Now we can show that every time we break down an X$d\pi$ network into a context and a smaller network, we can do the same in Core X$d\pi$, and vice versa.

Lemma 4.3.9 (Contextual Correspondence) *Let C and K respectively range over network contexts for X$d\pi$ and Core X$d\pi$.*

1. *For any C there is a K such that, for all $N \in \mathcal{N}$, $(\!|C[N]|\!) = K[(\!|N|\!)]$.*
2. *For any K there is a C such that, for all $N \in \mathcal{N}_{\mathrm{C}}$, $K[N] \equiv (\!|C[N']|\!)$, where $N' \in \mathcal{N}$ is such that $N \equiv (\!|N'|\!)$.*

Proof.

1. By a simple induction on the structure of $C[-]$.

2. Let $N = (D, P)$. Without loss of generality, we can assume that $K[-]$ has the form $(B \uplus -, (\nu\, \widetilde{c})(Q \mid -)$. In order for N, $K[-]$ and $K[N]$ to be well formed, it must be the case that $\mathit{dom}(P) \subseteq \mathit{dom}(D)$, $\mathit{dom}(Q) \subseteq \mathit{dom}(B)$ and $\mathit{dom}(B) \cap \mathit{dom}(D) = \emptyset$. By Lemma 4.3.8, there are N' and N'' such that $(\!|N'|\!) \equiv N$ and $(\!|N''|\!) \equiv (B, Q)$. By definition of the encoding, $C[-] = (\nu\, \widetilde{c})(N'' \mid -)$ is such that $(\!|C[N']|\!) \equiv K[N]$.

□

Reductions. We can now show that if the encoding of queries is homomorphic, then the encoding preserves every single reduction step. We will use this property to show that weak reductions are preserved.

Lemma 4.3.10 (Strong Operational Correspondence) *For any Xdπ network N, if $\{\!|-|\!\}$ is homomorphic then*

1. *if $N \longrightarrow M$ then $(\!|N|\!) \longrightarrow (\!|M|\!)$;*
2. *if $(\!|N|\!) \equiv N'$ and $N' \longrightarrow M'$ then there exists M such that $N \longrightarrow M$ and $(\!|M|\!) \equiv M'$.*

Proof.

1. By induction on the derivation of $N \longrightarrow M$. We show the most interesting cases.

   (Red Go) Suppose

   $$l\,[\,T \parallel Q \mid \mathsf{go}\, m.P\,] \mid m\,[\,S \parallel R\,] \;\longrightarrow\; l\,[\,T \parallel Q\,] \mid m\,[\,S \parallel R \mid P\,]$$

   By definition of the encoding, $(\!|N|\!) = N'$ where

   $$N' = (\{l \mapsto [\![T]\!]\} \uplus \{m \mapsto [\![S]\!]\}, \langle\!|Q|\!\rangle_l \mid l{\cdot}\mathsf{go}\, m.\langle\!|P|\!\rangle_m \mid \langle\!|R|\!\rangle_m)$$

   By (CRed Struct) and (CRed Go), $N' \longrightarrow M'$ where

   $$M' = (\{l \mapsto [\![T]\!]\} \uplus \{m \mapsto [\![S]\!]\}, \langle\!|Q|\!\rangle_l \mid \langle\!|P|\!\rangle_m \mid \langle\!|R|\!\rangle_m)$$

   By definition of the encoding, $(\!|M|\!) \equiv M'$.

   (Red Com) Suppose

   $$l\,[\,T \parallel \overline{c}\langle\widetilde{\pi}\sigma\rangle \mid c(\widetilde{\pi}).\boldsymbol{P} \mid Q\,] \;\longrightarrow\; l\,[\,T \parallel \boldsymbol{P}\sigma \mid Q\,]$$

   By definition of the encoding, $(\!|N|\!) = N'$ where

   $$N' = (\{l \mapsto [\![T]\!]\}, \overline{l{\cdot}c}\langle\widetilde{\pi}[\![\sigma]\!]\rangle \mid l{\cdot}c(\widetilde{\pi}).\langle\!|\boldsymbol{P}|\!\rangle_l \mid \langle\!|Q|\!\rangle_l)$$

   By (CRed Com), $N' \longrightarrow M'$ where

   $$M' = (\{l \mapsto [\![T]\!]\}, \langle\!|\boldsymbol{P}|\!\rangle_l[\![\sigma]\!] \mid \langle\!|Q|\!\rangle_l)$$

   By definition of the encoding and Lemma 4.3.4, $(\!|M|\!) \equiv M'$.

(Red Run) Suppose

$$l\,[\,T \parallel (\widetilde{\pi})\boldsymbol{P} \circ \langle \widetilde{\pi}\sigma \rangle \mid Q\,] \;\longrightarrow\; l\,[\,T \parallel \boldsymbol{P}\sigma \mid Q\,]$$

By definition of the encoding, $(\!|N|\!) = N'$ where

$$N' = (\{l \mapsto \llbracket T \rrbracket\}, (x, \widetilde{\pi})\langle\!|\boldsymbol{P}|\!\rangle_x \circ \langle l, \widetilde{\pi}\llbracket\sigma\rrbracket\rangle \mid \langle\!|Q|\!\rangle_l)$$

By (CRed Com), $N' \longrightarrow M'$ where

$$M' = (\{l \mapsto \llbracket T \rrbracket\}, \langle\!|\boldsymbol{P}|\!\rangle_x\{^l/_x\}\llbracket\sigma\rrbracket \mid \langle\!|Q|\!\rangle_l)$$

By applying the first substitution and by definition of the encoding,

$$M' = (\{l \mapsto \llbracket T \rrbracket\}, \langle\!|\boldsymbol{P}|\!\rangle_l\llbracket\sigma\rrbracket \mid \langle\!|Q|\!\rangle_l)$$

By definition of the encoding and Lemma 4.3.4, $(\!|M|\!) \equiv M'$.

(Red Request) Suppose $\mathfrak{E}(p, T) = (T', U_1 \mid \ldots \mid U_n \mid \varnothing)$ and

$$l\,[\,T \parallel \mathsf{req}_p\langle c \rangle \mid Q\,] \longrightarrow l\,[\,T' \parallel \overline{c}\langle \mathtt{r}[\,U_1\,] \mid \ldots \mid \mathtt{r}[\,U_n\,] \mid \varnothing \rangle \mid Q\,]$$

By definition of the encoding, $(\!|N|\!) = N'$ where

$$N' = (\{l \mapsto \llbracket T \rrbracket\}, l{\cdot}\mathsf{req}_{\{\!|p|\!\}}\langle c \rangle \mid \langle\!|Q|\!\rangle_l)$$

Since $\{\!|-|\!\}$ is homomorphic, $\mathfrak{E}(\{\!|p|\!\}, \llbracket T \rrbracket) = (\llbracket T' \rrbracket, \llbracket U_1 \mid \ldots \mid U_n \mid \varnothing \rrbracket)$. By (CRed Request), $N' \longrightarrow M'$ where

$$M' = (\{l \mapsto \llbracket T' \rrbracket\}, l{\cdot}\overline{c}\langle \llbracket \mathtt{r}[\,U_1\,] \mid \ldots \mid \mathtt{r}[\,U_n\,] \mid \varnothing \rrbracket \rangle \mid \langle\!|Q|\!\rangle_l)$$

By definition of the encoding, $(\!|M|\!) \equiv M'$.

(Red Par) Suppose $N = N_1 \mid M_1 \longrightarrow N_1' \mid M_1 = M$ because $N_1 \longrightarrow N_1'$. Let $C[-] = - \mid M_1$. By Lemma 4.3.9, $(\!|N|\!) = K[(\!|N_1|\!)]$ for some $K[-]$ chosen independently from N_1. By inductive hypothesis, $(\!|N_1|\!) \longrightarrow (\!|N_1'|\!)$. By (CRed Context), $K[(\!|N_1|\!)] \longrightarrow K[(\!|N_1'|\!)]$ where $K[-]$ is such that $(\!|C[N_1']|\!) = K[(\!|N_1'|\!)]$.

2. By induction on the derivation of $N' \longrightarrow M'$, reasoning similarly to the previous point. We show the most representative cases.

(CRed Request) Suppose $\mathfrak{E}(p, T) = (T', U_1 \mid \ldots \mid U_n \mid \varnothing)$ and

$$N' = (\{l \mapsto T\}, l{\cdot}\mathsf{req}_p\langle c \rangle \mid Q) \longrightarrow l\,[\,T' \parallel \overline{l{\cdot}c}\langle \mathtt{r}[\,U_1\,] \mid \ldots \mid \mathtt{r}[\,U_n\,] \mid \varnothing \rangle \mid Q\,] = M'$$

Since N' must be well formed, $\mathit{dom}(Q) \subseteq \{l\}$. By syntactical reasoning on the definition of the encoding, since $N' \equiv (\!|N|\!)$, it must be the case that $N = l\,[\,T_1 \parallel \mathsf{req}_{p_1}\langle c \rangle \mid Q_1\,]$ where $\llbracket T_1 \rrbracket = T$, $\{\!|p_1|\!\} = p$ and $\langle\!|Q_1|\!\rangle_l = Q$. Since $\{\!|-|\!\}$ is homomorphic, $\mathfrak{E}(p_1, T_1) = (T_1', U_1' \mid \ldots \mid U_n' \mid \varnothing)$ where $\llbracket T_1' \rrbracket = T'$ and $\llbracket U_1' \mid \ldots \mid U_n' \mid \varnothing \rrbracket = U_1 \mid \ldots \mid U_n \mid \varnothing$. By (Red Request), $N \longrightarrow M$ where

$$M = l\,[\,T_1' \parallel \overline{c}\langle \mathtt{r}[\,U_1'\,] \mid \ldots \mid \mathtt{r}[\,U_n'\,] \mid \varnothing \rangle \mid Q\,]$$

By definition of the encoding, $(\!|M|\!) = M'$.

(CRed Context) Suppose $N' = K[N''] \longrightarrow K[M''] = M'$ because $N'' \longrightarrow M''$. By Lemma 4.3.9, there exist C and N_1 such that $K[N''] \equiv (\!|C[N_1]|\!)$ and $(\!|N_1|\!) \equiv N''$. By inductive hypothesis, $N_1 \longrightarrow M_1$ for some M_1 such that $(\!|M_1|\!) \equiv M''$. By a simple induction on the structure of C, using rules (Red Par), (Red Res) and the premise $N_1 \longrightarrow M_1$, we get $C[N_1] \longrightarrow C[M_1]$. By Lemma 4.3.9, $(\!|C[M_1]|\!) \equiv K[M'']$.

□

The property below is crucial to compare the reduction congruences in the two languages.

Lemma 4.3.11 (Weak Operational Correspondence) *Given any Xdπ network N, (i) if $N \longrightarrow^{*} M$ then $(\!|N|\!) \longrightarrow^{*} (\!|M|\!)$; (ii) if $(\!|N|\!) \equiv N'$ and $N' \longrightarrow^{*} M'$ then there exist an M such that $N \longrightarrow^{*} M$ and $(\!|M|\!) \equiv M'$.*

Proof. Both cases follow by induction on the number of reduction steps and Lemma 4.3.10. The second case uses also Lemma 4.3.8. □

Observations. The last remaining step, before comparing the equivalences, consists of showing that the encoding preserves the observables. To do this, we find it convenient to define when an encoding *maps* precisely each observable from a source set into one in a target set.

Definition 4.3.12 (Observation Mapping) *Let $\mathcal{O}$ and $\mathcal{O}'$ be arbitrary sets of observables for Xdπ and Core Xdπ respectively. We say that the encoding $(\!|-|\!)$* maps *$\mathcal{O}$ to $\mathcal{O}'$ if*

1. *for all $\alpha \in \mathcal{O}$ there is $\beta \in \mathcal{O}'$ such that given any $N \in \mathcal{N}$, $N \downarrow_\alpha \iff (\!|N|\!) \downarrow_\beta$;*
2. *for all $\beta \in \mathcal{O}'$ there is $\alpha \in \mathcal{O}$ such that given any $N \in \mathcal{N}$, $(\!|N|\!) \downarrow_\beta \iff N \downarrow_\alpha$.*

Independently from the choice of a query language, $(\!|-|\!)$ maps both tree and channel observables from X$d\pi$ to Core X$d\pi$. Moreover, if the encoding of queries is observation-injective, then $(\!|-|\!)$ maps also the request observables.

Proposition 4.3.13 (Observation Mapping) *The encoding $(\!|-|\!)$*

1. *maps $\mathcal{O}_{\mathrm{t}}$ to $\mathcal{O}_{\mathrm{t^C}}$;*
2. *maps $\mathcal{O}_{\mathrm{c}}$ to $\mathcal{O}_{\mathrm{c^C}}$;*
3. *if $\{\!|-|\!\}$ is observation-injective, maps $\mathcal{O}_{\mathrm{r}}$ to $\mathcal{O}_{\mathrm{r^C}}$.*

Proof.

1. We show point 1 of Definition 4.3.12. Suppose $N \downarrow_{l \cdot T}$. By definition of $\mathcal{O}_{\mathrm{t}}$, $N \equiv C[l\,[\,T' \,\|\, P\,]]$ and $T \eqsim T'$. By Lemma 4.3.6 and definition of the encoding, $(\!|N|\!) \equiv (\{l \mapsto \llbracket T' \rrbracket\} \uplus D, \langle\!|P|\!\rangle_l \,|\, Q)$ where $\llbracket T' \rrbracket \eqsim \llbracket T \rrbracket$. By Lemma 4.3.7, $T \eqsim$

$[\![T]\!]$. By definition of $\mathcal{O}_{\mathrm{tC}}$, $(\![N]\!) \downarrow_{l\cdot[\![T]\!]}$. Suppose now $(\![N]\!) \downarrow_{l\cdot[\![T]\!]}$. By definition of $\mathcal{O}_{\mathrm{tC}}$, Lemma 4.3.6 and definition of the encoding, $(\![N]\!) \equiv (\{l \mapsto [\![T']\!]\} \uplus D, (\!|P|\!)_l \mid Q)$ where $[\![T']\!] \doteq [\![T]\!]$ and $N \equiv C[l\,[\,T' \parallel P\,]]$. By Lemma 4.3.7, $T' \doteq [\![T']\!]$. By definition of $\mathcal{O}_{\mathrm{t}}$, $N \downarrow_{l\cdot T}$. The case for point 2 follows a similar structure.

2. Suppose $N \downarrow_{l\cdot a}$. By definition of $\mathcal{O}_{\mathrm{c}}$, $N \equiv C[l\,[\,T \parallel \overline{a}\langle\widetilde{v}\rangle \mid P\,]]$ where $a \in \mathit{fn}(N)$. By Lemma 4.3.6 and definition of the encoding, $(\![N]\!) \equiv (\{l \mapsto [\![T]\!]\} \uplus D, \overline{l\cdot a}\langle[\![\widetilde{v}]\!]\rangle \mid (\!|P|\!)_l \mid Q)$. By definition of $\mathcal{O}_{\mathrm{cC}}$, $(\![N]\!) \downarrow_{l\cdot a}$. Suppose now $(\![N]\!) \downarrow_{l\cdot a}$. By definition of $\mathcal{O}_{\mathrm{cC}}$, Lemma 4.3.6 and definition of the encoding, $(\![N]\!) \equiv (\{l \mapsto [\![T]\!]\} \uplus D, \overline{l\cdot a}\langle[\![\widetilde{v}]\!]\rangle \mid (\!|P|\!)_l \mid Q)$ where $a \in \mathit{fn}(N)$ and $N \equiv C[l\,[\,T \parallel \overline{a}\langle\widetilde{v}\rangle \mid P\,]]$. By definition of $\mathcal{O}_{\mathrm{c}}$, $N \downarrow_{l\cdot a}$. The other direction follows a similar structure.

3. Suppose $N \downarrow_{l\cdot p}$. By definition of $\mathcal{O}_{\mathrm{r}}$, $N \equiv C[l\,[\,T \parallel \mathsf{req}_p\langle c\rangle \mid P\,]]$. By Lemma 4.3.6 and definition of the encoding, $(\![N]\!) \equiv (\{l \mapsto [\![T]\!]\} \uplus D, l\cdot\mathsf{req}_{\{\![p]\!\}}\langle c\rangle \mid (\!|P|\!)_l \mid Q)$. By definition of $\mathcal{O}_{\mathrm{rC}}$, $(\![N]\!) \downarrow_{l\cdot\{\![p]\!\}}$. Suppose now $(\![N]\!) \downarrow_{l\cdot\{\![p]\!\}}$. By definition of $\mathcal{O}_{\mathrm{rC}}$, Lemma 4.3.6 and definition of the encoding, $(\![N]\!) \equiv (\{l \mapsto [\![T]\!]\} \uplus D, l\cdot\mathsf{req}_{\{\![p]\!\}}\langle c\rangle \mid (\!|P|\!)_l \mid Q)$ and $N \equiv C[l\,[\,T \parallel \mathsf{req}_q\langle c\rangle \mid P\,]]$ for some q such that $\{\![q]\!\} = \{\![p]\!\}$. By definition of $\mathcal{O}_{\mathrm{r}}$, $N \downarrow_{l\cdot q}$. By observation-injectivity of $\{\![-]\!\}$, since $l\cdot\{\![p]\!\} = l\cdot\{\![q]\!\}$, we have that $l\cdot q = l\cdot p$, hence $N \downarrow_{l\cdot p}$. The other direction follows a similar structure.

□

In general, if an encoding is observation-mapping and it preserves weak reduction, then it preserves the observables up-to structural congruence.

Lemma 4.3.14 (Observational Correspondence) *Let $\mathcal{O}$ and $\mathcal{O}'$ be arbitrary sets of observables such that $(\![-]\!)$ maps $\mathcal{O}$ to $\mathcal{O}'$, and $\mathcal{O}'$ is defined up-to structural congruence (Definition 4.2.2).*

1. *For all $\alpha \in \mathcal{O}$ there is $\beta \in \mathcal{O}'$ such that given any $N \in \mathcal{N}$, $N \Downarrow_\alpha \iff (\![N]\!) \Downarrow_\beta$.*
2. *For all $\beta \in \mathcal{O}'$ there is $\alpha \in \mathcal{O}$ such that given any $N \in \mathcal{N}$, $(\![N]\!) \Downarrow_\beta \iff N \Downarrow_\alpha$.*

Proof. Follows by Definition 4.3.12 and by using the fact that $\mathcal{O}'$ is defined up-to structural congruence together with Lemma 4.3.11. □

Full abstraction. We can now show the main result of this section: if $(\![-]\!)$ preserves the set of observables and $\{\![-]\!\}$ is homomorphic, then $(\![-]\!)$ preserves the corresponding induced reduction congruence.

Theorem 4.3.15 (Generalized Full Abstraction) *Let $\mathcal{O}$ and $\mathcal{O}'$ be arbitrary sets of observables up-to $\equiv$ for Xdπ and Core Xdπ respectively, such that $(\![-]\!)$ maps $\mathcal{O}$ to $\mathcal{O}'$ and $\{\![-]\!\}$ is homomorphic. For any $N, M \in \mathcal{N}$, $N \simeq_{\mathcal{O}} M$ if and only if $(\![N]\!) \simeq_{\mathcal{O}'} (\![M]\!)$.*

Proof. Follows from Lemma 4.3.9, Lemma 4.3.11 and Lemma 4.3.14. □

Theorem 4.3.15 applies straightforwardly to the reduction congruences defined in Section 4.2, justifying the use of Core X$d\pi$ as an alternative setting in which to study equivalences for X$d\pi$.

Corollary 4.3.16 (Full Abstraction) *Let* $\{\!|-|\!\}$ *be the encoding for* Sam *queries given in Definition 4.3.3. For any* $N, M \in \mathcal{N}$,

- $N \simeq_{\mathrm{t}} M$ *if and only if* $(\!|N|\!) \simeq_{\mathrm{t^C}} (\!|M|\!)$;
- $N \simeq_{\mathrm{c}} M$ *if and only if* $(\!|N|\!) \simeq_{\mathrm{c^C}} (\!|M|\!)$;
- $N \simeq_{\mathrm{r}} M$ *if and only if* $(\!|N|\!) \simeq_{\mathrm{r^C}} (\!|M|\!)$.

Proof. Both three cases follow from Proposition 4.3.5, Proposition 4.3.13 and Theorem 4.3.15. □

4.4 Process equivalences

We now define process equivalences for Core X$d\pi$. These equivalences depend on the locations present in the network. Consider replacing the definition of a service at location l, which uses only local data, with an equivalent one depending on data from another location m. If location m is connected, then the behaviour of the services is the same. On the other hand, if location m is not connected, the behaviour of the services is different. With network equivalences, the connected locations are those in the domain of the store. With process equivalences, we must state explicitly the locations which we assume to be part of the network. As a consequence, process equivalence is indexed by a *domain* (a set of locations) Λ.

A Core X$d\pi$ process can be seen as a partial specification of a network, describing only some of the processes running in some of the locations. This point of view is useful for reasoning about replacing components which are part of some distributed data-exchange protocol. Accordingly, we say that two processes P and Q are equivalent with respect to a domain Λ if all the networks containing *at least* the locations in Λ and either P or Q, are equivalent.

Besides comparing partial network specifications, process equivalences can be useful for example to replace optimized pieces of code inside a specific process. For that purpose, we need a more general class of process contexts.

Definition 4.4.1 (Full Contexts) Full process contexts $\mathcal{K}_{\mathrm{f}}$ *are the terms generated by*

$$C ::= \ - \ \mid C \mid \boldsymbol{P} \mid \boldsymbol{P} \mid C \mid (\nu\, c)C \mid \boldsymbol{l}{\cdot}\boldsymbol{a}(\widetilde{\pi}).C \mid !\boldsymbol{l}{\cdot}\boldsymbol{a}(\widetilde{\pi}).C \mid \boldsymbol{l}{\cdot}\mathsf{go}\,\boldsymbol{m}.C$$

Unless we specify otherwise, from now on we use $C[-]$ to denote full contexts.

Figure 4.7: Notation for asynchronous processes

(Forwarder)	$\boldsymbol{l}\cdot\mathsf{FW}(\boldsymbol{a},\boldsymbol{b},\widetilde{\pi}) \stackrel{\text{def}}{=} \boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\overline{\boldsymbol{l}\cdot\boldsymbol{b}}\langle\widetilde{\pi}\rangle$
(Equator)	$\boldsymbol{l}\cdot\mathsf{EQ}(\boldsymbol{a},\boldsymbol{b},\widetilde{\pi}) \stackrel{\text{def}}{=} !\boldsymbol{l}\cdot\mathsf{FW}(\boldsymbol{a},\boldsymbol{b},\widetilde{\pi}) \mid !\boldsymbol{l}\cdot\mathsf{FW}(\boldsymbol{b},\boldsymbol{a},\widetilde{\pi})$
(Distributed Forwarder)	$\boldsymbol{l}\cdot\mathsf{dFW}(\boldsymbol{a},\boldsymbol{m},\boldsymbol{b},\widetilde{\pi}) \stackrel{\text{def}}{=} \boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\overline{\boldsymbol{m}\cdot\boldsymbol{b}}\langle\widetilde{\pi}\rangle$
(Distributed Equator)	$\mathsf{dEQ}(\boldsymbol{l},\boldsymbol{a},\boldsymbol{m},\boldsymbol{b},\widetilde{\pi}) \stackrel{\text{def}}{=} !\boldsymbol{l}\cdot\mathsf{dFW}(\boldsymbol{a},\boldsymbol{m},\boldsymbol{b},\widetilde{\pi}) \mid !\boldsymbol{m}\cdot\mathsf{dFW}(\boldsymbol{b},\boldsymbol{l},\boldsymbol{a},\widetilde{\pi})$

Definition 4.4.2 (Induced Domain Congruence) *Given a set of location names Λ and an induced reduction congruence $\simeq_{\mathcal{O}}$, we define the* induced domain congruence $\sim_{\mathcal{O}}^{\Lambda}$ *on processes by*

$$\sim_{\mathcal{O}}^{\Lambda} = \{(\boldsymbol{P},\boldsymbol{Q}) \;:\; \forall D, C[-].\ \Lambda \subseteq \mathit{dom}(D) \Longrightarrow (D, C[\boldsymbol{P}]) \simeq_{\mathcal{O}} (D, C[\boldsymbol{Q}])\}$$

where each $C[-]$ is closing for both $\boldsymbol{P}$ and $\boldsymbol{Q}$.

In the rest of the thesis, we will give a prominent role to the domain congruence induced by request observables (*request congruence* for short), as in general it implies the ones induced by channel and tree observables.

Domain congruences are monotonic: the larger the set of locations which we assume to be part of the network, the larger the number of processes which we can equate using a domain congruence.

Observation 4.4.3 (Monotonicity) *Given any induced reduction congruence $\simeq_{\mathcal{O}}$, if $\Lambda \subseteq \Lambda'$ then $\sim_{\mathcal{O}}^{\Lambda} \subseteq \sim_{\mathcal{O}}^{\Lambda'}$.*

Proof. Follows easily by Definition 4.4.2. □

Remark 4.4.4 (Asynchronous Laws) *Core Xdπ is an extension of the asynchronous π-calculus, so we consider some equational laws inspired by the latter. Consider the process definitions given in Figure 4.7. The* asynchrony law*, stating that the presence of a communication buffer cannot be observed, holds also in Core Xdπ (see Section 5.2 for a proof):*

$$!\boldsymbol{l}\cdot\mathsf{FW}(\boldsymbol{a},\boldsymbol{a},\widetilde{\pi}) \sim_r^{\Lambda} \mathbf{0}$$

The law stating that two channels $\boldsymbol{a}$ and $\boldsymbol{b}$ cannot be distinguished if they are part of the same equator does not hold. Fore example,

$$\boldsymbol{l}\cdot\mathsf{EQ}(\boldsymbol{a},\boldsymbol{b},\widetilde{\pi}) \mid \overline{\boldsymbol{l}\cdot\boldsymbol{c}}\langle\boldsymbol{a}\rangle \not\sim_r^{\Lambda} \boldsymbol{l}\cdot\mathsf{EQ}(\boldsymbol{a},\boldsymbol{b},\widetilde{\pi}) \mid \overline{\boldsymbol{l}\cdot\boldsymbol{c}}\langle\boldsymbol{b}\rangle$$

because a context could intercept the channel name $\boldsymbol{a}$ and use it in some fresh location m where $\boldsymbol{a}$ and $\boldsymbol{b}$ are not equated. We have instead a new law about equating located channels across different locations:

$$\mathsf{dEQ}(l,\boldsymbol{a},m,\boldsymbol{b},\widetilde{\pi}) \mid \overline{l\cdot\boldsymbol{a}}\langle\widetilde{\pi}\sigma\rangle \sim_r^{\{l,m\}} \mathsf{dEQ}(l,\boldsymbol{a},m,\boldsymbol{b},\widetilde{\pi}) \mid \overline{m\cdot\boldsymbol{b}}\langle\widetilde{\pi}\sigma\rangle$$

This law could be useful to show that we can replicate Web services (improving efficiency) without the clients needing to be aware of the change.

Due to the several explicit (and implicit) universal quantifications involved in Definition 4.4.2, it is very difficult to show directly that two processes are domain congruent. For this reason, in Chapter 5 we will introduce a proof method which does not require closure under contexts and which entails domain congruence.

Chapter 5

Bisimilarity

*Domain congruence is hard to use in practice, because it requires closure under both store and process contexts. In this chapter, we define a coinductive equivalence relation (*bisimilarity*), which does not quantify over contexts and which entails domain congruence. In Section 5.1, we define a labelled transition system for Core Xdπ inspired by the work of Sangiorgi [69], and Jeffrey and Rathke [46]. In Section 5.2, we define* domain bisimilarity*, our coinductive equivalence, and discuss its main properties. Section 5.3 contains all the formal results with proofs.*

5.1 Labelled transition system

A typical proof that processes are bisimilar involves a universal quantification over labelled transitions. Since Core Xdπ values include scripts, and labels typically include values, we risk falling back to quantifying over processes. Following the approach of [69, 46], we avoid this problem by translating messages containing scripts into ones where each script is replaced by a *trigger name* (a first-order value), and by placing in parallel to the process being analyzed some *definitions* associating to each trigger name the code of the corresponding script. By including these definitions in the code, we are able to analyze also the interaction between scripts and their contexts.

Configurations. We introduce *configurations*, which are processes extended with the trigger names and definitions mentioned above. The formal syntax is given in Figure 5.1. Note that $\boldsymbol{A}$ denotes now a script, a variable or a trigger name, hence processes can syntactically contain triggers. Nonetheless, scripts and queries are not allowed to contain triggers. In fact, trigger names and definitions are merely intermediate terms arising during the analysis of the transition of a process, and are not meant to be part of the user syntax. For a configuration K to be well-formed, no two definitions in K can have the same trigger name (predicate $\mathit{unique}(K)$). As a convention, we let Θ, Ω and Φ range on groups of definitions. Note also that two groups of definitions $\Theta^{\widetilde{k}}$ and $\Theta^{\widetilde{j}}$ identified by the same name but by different vectors of triggers can in principle be arbitrarily different: it is an important syntactic convention which helps to simplify the notation, and is often used in the rest of the chapter. In Figure A.5 and Figure A.6, we extend $\equiv$, fv and fn to configurations. In Figure 5.3 we extend the function dom of Figure A.2 to configurations, and we define

Figure 5.1: Syntax: configurations

$\boldsymbol{K}$ **::=**		configuration terms
	$\boldsymbol{P}$	process terms (built as for Core X$d\pi$)
	$\boldsymbol{K} \mid \boldsymbol{K}$	parallel composition
	$(\nu\, c)\boldsymbol{K}$	restriction
	$\langle k \Leftarrow A \rangle$	definition for trigger name k with script A
$\boldsymbol{A}$ **::=**	$A \mid x \mid k$	script or variable or trigger name

$h, i, j, k \in \mathcal{Y}$ $\quad (\mathcal{Y} \cap \mathcal{C}_{\mathrm{p}} = \emptyset,\ \mathcal{Y}$ countably infinite) (Trigger Names)

$K, L \in \mathcal{W} \stackrel{\text{def}}{=} \{\boldsymbol{K} \ : \ \mathit{fv}(\boldsymbol{K}) = \emptyset,\ \mathit{unique}(\boldsymbol{K})\}$ (Configurations)

$A \in \mathcal{A}_{\mathcal{W}} \stackrel{\text{def}}{=} \{A \ : \ A \in \mathcal{A}_{\mathrm{C}},\ \mathit{triggers}(A) = \emptyset\}$ (Scripts)

$C_{\mathcal{W}}[-]$ **::=** $- \mid C_{\mathcal{W}}[-] \mid K \mid K \mid C_{\mathcal{W}}[-] \mid (\nu\, c)C_{\mathcal{W}}[-]$ (Configuration Contexts)

The function *triggers* and the predicate *unique* are defined in Figure 5.2.
Apart from the redefinition of $\boldsymbol{A}$, A and p (see below), the grammars for trees, processes and values are the same as for Core X$d\pi$.

For trigger definitions, we adopt the following notation:

$\Theta^{\widetilde{k}} \stackrel{\text{def}}{=} \langle k_1 \Leftarrow A_1 \rangle \mid \ldots \mid \langle k_n \Leftarrow A_n \rangle$, where all k_i are distinct, $n \geq 0$;
$\Theta^{\widetilde{k} \curvearrowleft \widetilde{j}} \stackrel{\text{def}}{=} \Theta^{\widetilde{k}}\{{}^{\widetilde{j}}/_{\widetilde{k}}\}$, when $\{{}^{\widetilde{j}}/_{\widetilde{k}}\}$ is defined;
$\Theta \stackrel{\text{def}}{=} \Theta^{\widetilde{k}}$, when $\widetilde{k}$ is not important.
$t^{\langle k \Leftarrow A \rangle} \stackrel{\text{def}}{=} t\{{}^{A}/_{k}\}$, for any term t, and similarly for $t^{\Theta^{\widetilde{k}}}$.

Figure 5.2: Function *triggers* and predicate *unique*.

$$\mathit{triggers}(t) = \mathit{fn}(t) \cap \mathcal{Y}$$

$$\frac{\mathit{unique}(\boldsymbol{K}) \quad \mathit{unique}(\boldsymbol{K}') \quad \mathit{dfs}(\boldsymbol{K}) \cap \mathit{dfs}(\boldsymbol{K}') = \emptyset}{\mathit{unique}(\boldsymbol{K} \mid \boldsymbol{K}')} \qquad \frac{\mathit{unique}(\boldsymbol{K})}{\mathit{unique}((\nu\, c)\boldsymbol{K})} \qquad \mathit{unique}(\langle k \Leftarrow A \rangle)$$

The function *dfs*, returning the triggers defined by a configuration, is given by

$$\mathit{dfs}(\boldsymbol{K} \mid \boldsymbol{K}') = \mathit{dfs}(\boldsymbol{K}) \cup \mathit{dfs}(\boldsymbol{K}') \quad \mathit{dfs}((\nu\, c)\boldsymbol{K}) = \mathit{dfs}(\boldsymbol{K}) \quad \mathit{dfs}(\langle k \Leftarrow A \rangle) = \{k\}$$

a function *scripts* returning the scripts present in a piece of data.

Queries. Queries used for updating can mention constant data, which may contain scripts. We assume two functions, *scripts* and *triggers*, which given a query return respectively the set of scripts and triggers it contains. The only condition

Figure 5.3: Functions *dom* and *scripts* for configurations

$$\mathit{dom}(\boldsymbol{K} \mid \boldsymbol{K}') = \mathit{dom}(\boldsymbol{K}) \cup \mathit{dom}(\boldsymbol{K}') \quad \mathit{dom}((\nu\, c)\boldsymbol{K}) = \mathit{dom}(\boldsymbol{K}) \quad \mathit{dom}(\langle k \Leftarrow A\rangle) = \emptyset$$

$$\begin{aligned}
\mathit{scripts}(v, \widetilde{v}) &= \mathit{scripts}(v) \cup \mathit{scripts}(\widetilde{v}) & \mathit{scripts}(\{{}^{v}/_{x}\}) &= \mathit{scripts}(v)\\
\mathit{scripts}(E \shortmid T) &= \mathit{scripts}(E) \cup \mathit{scripts}(T) & \mathit{scripts}(\varnothing) &= \emptyset\\
\mathit{scripts}(\mathtt{a}[\,V\,]) &= \mathit{scripts}(V) & \mathit{scripts}(\langle A\rangle) &= \{A\}\\
\mathit{scripts}(A) &= \{A\} & \mathit{scripts}(c) = \mathit{scripts}(\mathtt{c}) &= \emptyset\\
\mathit{scripts}(p@l) &= \mathit{scripts}(p) & \mathit{scripts}(l) &= \emptyset
\end{aligned}$$

Assumption: *scripts* on queries is given as part of the query language definition.

that we need to impose on query evaluation consists of it not being dependent on the particular structure of scripts. In other words, if we replace a script in a query with a trigger name, then the result of the query should be equivalent up to substitution of the script for the trigger. Moreover, any script returned by the query must occur in the input tree or in the query itself. The condition is formalized below.

Definition 5.1.1 (Script Independence) *Let $\mathfrak{L} = (\mathcal{Q}, \mathit{fv}, \mathfrak{E})$ be an arbitrary query language, let p, T be such that $\mathfrak{E}(p, T) = (S, L)$, and let p_0, T_0 be their first-order versions, such that $\mathit{scripts}(p_0) = \mathit{scripts}(T_0) = \emptyset$ and $p = p_0^{\widetilde{\Theta^j}}, T = T_0^{\widetilde{\Omega^k}}$ for some $\widetilde{\Theta^j}, \widetilde{\Omega^k}$.*

The query language $\mathfrak{L}$ is script independent *if for all $\widetilde{\Theta^j}, \widetilde{\Omega^k}$ there exist $\widetilde{\Theta^h}$ and $\widetilde{\Omega^i}$ such that*

- *query evaluation does not depend on the structure of scripts: there are S_0, L_0 with $\mathit{scripts}(S_0) = \mathit{scripts}(L_0) = \emptyset$ such that $\mathfrak{E}(p_0^{\widetilde{\Theta^j}}, T_0^{\widetilde{\Omega^k}}) = (S_0^{\widetilde{\Theta^h}}, L_0^{\widetilde{\Omega^i}})$*
- *no new scripts are introduced: for any definition $\langle k \Leftarrow A\rangle$ occurring in $\widetilde{\Theta^h}$ or $\widetilde{\Omega^i}$ there must be a definition $\langle k' \Leftarrow A\rangle$ occurring in $\widetilde{\Theta^j}$ or $\widetilde{\Omega^k}$.*

Extracting scripts from values. Our strategy consists of translating values containing scripts into values containing trigger names only, extracting at the same time the corresponding definitions. For that purpose, we define in Figure 5.4 an extraction relation $\mathfrak{X}$ which applies to Core X$d\pi$ data and stores, and returns the corresponding first-order terms and the definitions extracted.

The definition of $\mathfrak{X}$ is straightforward. The only points worth noting are that the premises of the rules for tuples, tree and store composition make sure that the trigger names remain disjoint, and that the rule for scripts replaces a script with a trigger and records the corresponding definition. The rule for queries invokes a specialized extraction relation $\mathfrak{X}_{\mathcal{Q}}$ which depends on the query language. $\mathfrak{X}_{\mathcal{Q}}$ can behave similarly to $\mathfrak{X}$, relating each query with its first-order version and the corresponding definitions, or can behave differently (for example being the identity function on queries and the

Figure 5.4: Extraction relation

$$\mathfrak{X}(l) = (l; \mathbf{0}) \qquad \frac{\mathfrak{X}(\boldsymbol{p}) = (\boldsymbol{p}'; \Theta)}{\mathfrak{X}(\boldsymbol{p}@\boldsymbol{l}) = (\boldsymbol{p}'@\boldsymbol{l}; \Theta)}$$

$$\mathfrak{X}(p) = \mathfrak{X}_{\mathcal{Q}}(p) \qquad \mathfrak{X}(\varnothing) = (\varnothing; \mathbf{0})$$

$$\mathfrak{X}(A) = (k; \langle k \Leftarrow A \rangle) \qquad \mathfrak{X}(x) = (x; \mathbf{0})$$

$$\mathfrak{X}(c) = (c; \mathbf{0}) \qquad \mathfrak{X}(k) = (k; \mathbf{0})$$

$$\frac{\mathfrak{X}(\boldsymbol{E}) = (\boldsymbol{E}'; \Theta^{\widetilde{k}}) \qquad \mathfrak{X}(\boldsymbol{T}) = (\boldsymbol{T}'; \Omega^{\widetilde{h}}) \qquad (\{\widetilde{k}, \widetilde{h}\} \cap \mathit{fn}(\boldsymbol{E}\mathbin{\imath}\boldsymbol{T})) \cup (\{\widetilde{k}\} \cap \{\widetilde{h}\}) = \emptyset}{\mathfrak{X}(\boldsymbol{E}\mathbin{\imath}\boldsymbol{T}) = (\boldsymbol{E}'\mathbin{\imath}\boldsymbol{T}'; \Theta^{\widetilde{k}} \,|\, \Omega^{\widetilde{h}})} \qquad \frac{\mathfrak{X}(\boldsymbol{A}) = (\boldsymbol{v}; \Theta)}{\mathfrak{X}(\langle \boldsymbol{A} \rangle) = (\langle \boldsymbol{v} \rangle; \Theta)}$$

$$\frac{\mathfrak{X}(\boldsymbol{v}) = (\boldsymbol{v}'; \Theta^{\widetilde{k}}) \qquad \mathfrak{X}(\widetilde{\boldsymbol{v}}) = (\widetilde{\boldsymbol{v}'}; \Omega^{\widetilde{h}}) \qquad (\{\widetilde{k}, \widetilde{h}\} \cap \mathit{fn}(\boldsymbol{v}, \widetilde{\boldsymbol{v}})) \cup (\{\widetilde{k}\} \cap \{\widetilde{h}\}) = \emptyset}{\mathfrak{X}(\boldsymbol{v}, \widetilde{\boldsymbol{v}}) = (\boldsymbol{v}', \widetilde{\boldsymbol{v}'}; \Theta^{\widetilde{k}} \,|\, \Omega^{\widetilde{h}})} \qquad \frac{\mathfrak{X}(\boldsymbol{V}) = (\boldsymbol{V}'; \Theta)}{\mathfrak{X}(\mathtt{a}[\,\boldsymbol{V}\,]) = (\mathtt{a}[\,\boldsymbol{V}'\,]; \Theta)}$$

$$\frac{\mathfrak{X}(T) = (T'; \Theta^{\widetilde{k}}) \qquad \mathfrak{X}(D) = (D'; \Omega^{\widetilde{h}}) \qquad (\{\widetilde{k}, \widetilde{h}\} \cap \mathit{fn}(T, D)) \cup (\{\widetilde{k}\} \cap \{\widetilde{h}\}) = \emptyset}{\mathfrak{X}(\{l \mapsto T\} \uplus D) = (\{l \mapsto T'\} \uplus D'; \Theta^{\widetilde{k}} \,|\, \Omega^{\widetilde{h}})} \qquad \mathfrak{X}(\emptyset) = (\emptyset; \mathbf{0})$$

Notation: in this table c ranges over $\mathcal{C}_{\mathrm{p}} \cup \mathcal{C}_{\mathrm{s}}$.
The relation $\mathfrak{X}_{\mathcal{Q}}$ is specified in Definition 5.1.2.

constant 0 on configurations), as long as it satisfies the basic properties requested by the definition given below.

Definition 5.1.2 (Query Extraction) *The relation $\mathfrak{X}_{\mathcal{Q}}$ can be any subset of $\mathcal{Q}_{\mathrm{C}} \times \mathcal{Q}_{\mathrm{C}} \times \mathcal{W}$ satisfying the condition that if $\mathfrak{X}_{\mathcal{Q}}(p) = (p'; K)$ then*

1. *K are well-formed definitions: $K = \Theta^{\widetilde{k}}$;*
2. *trigger names can be extended as long as there are no clashes: $\mathit{triggers}(p') = \mathit{triggers}(p) \cup \{\widetilde{k}\}$ and $\mathit{triggers}(p) \cap \{\widetilde{k}\} = \emptyset$;*
3. *the new trigger names are defined up-to renaming: for all $\widetilde{j}$ distinct from $\mathit{triggers}(p)$, $\mathfrak{X}_{\mathcal{Q}}(p) = (p'\{\widetilde{j}/\widetilde{k}\}; \Theta^{\widetilde{k} \curvearrowleft \widetilde{j}})$;*
4. *substitution is the inverse of extraction: if $\mathfrak{X}_{\mathcal{Q}}(p) = (p'; \Theta)$ then $p = p'^{\Theta}$.*

Under the assumption (that we adopt henceforth) that $\mathfrak{X}_{\mathcal{Q}}$ respects Definition 5.1.2, the effects of relation $\mathfrak{X}$ can be reversed by replacing, in the extracted first-order term, the new trigger names by the corresponding definitions.

Observation 5.1.3 (Extraction) *For any given term t, if $\mathfrak{X}(t) = (t'; \Theta)$ then $t = t'^{\Theta}$.*

Figure 5.5: Labels for the transition system

α_l **::=**		transition labels
	$(\widetilde{a},\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v}\rangle$	output of $\widetilde{v}$ on c at l, extruding $\widetilde{a},\widetilde{k}$
	$l{\cdot}c(\widetilde{v})$	input of $\widetilde{v}$ on c at l
	$l{\cdot}\tau$	internal reduction at l
	$(\widetilde{k})l{\cdot}\mathsf{req}\langle p\rangle(T)$	request p at l extruding $\widetilde{k}$, obtaining result T
	$l{\cdot}k(\widetilde{v})$	run the script defined by trigger k with parameters $l,\widetilde{v}$
	$(\widetilde{a},\widetilde{k})\overline{l{\cdot}j}\langle\widetilde{v}\rangle$	assume running the script defined by trigger j with parameters $l,\widetilde{v}$, extruding $\widetilde{a},\widetilde{k}$

Convention: in this figure, c ranges over $\mathcal{C}_\mathrm{p}\cup\mathcal{C}_\mathrm{s}$, and t ranges over $\mathcal{C}_\mathrm{p}\cup\mathcal{C}_\mathrm{s}\cup\mathcal{Y}$.

labels α_l	names n	bound names bn
$(\widetilde{a},\widetilde{k})\overline{l{\cdot}t}\langle\widetilde{v}\rangle$	$\{\widetilde{a}\}\cup\{\widetilde{k}\}\cup\{t\}\cup fn(\widetilde{v})$	$\{\widetilde{a}\}\cup\{\widetilde{k}\}$
$l{\cdot}t(\widetilde{v})$	$\{t\}\cup fn(\widetilde{v})$	$\emptyset$
$l{\cdot}\tau$	$\emptyset$	$\emptyset$
$(\widetilde{k})l{\cdot}\mathsf{req}\langle p\rangle(T)$	$\{\widetilde{k}\}\cup \mathit{triggers}(p)\cup fn(T)$	$\{\widetilde{k}\}$

$$fn(\alpha_l) = n(\alpha_l)\setminus bn(\alpha_l)$$

Proof. By induction on the derivation of $\mathfrak{X}(t) = (t';\Theta)$. □

Labelled transition system. The labels of the transition system record what kind of interaction with the external environment is necessary for a configuration to evolve into another. Labels, along with the notions of their names, free names and bound names, are defined in Figure 5.5. Each label, including the one for internal reduction, shows explicitly the location where interaction takes place. By using appropriate conditions on the function *scripts* in the rules of the labelled transition system (*lts* for short), we will guarantee that labels are first-order, as planned. The formal definition of the lts is given in Figure 5.6. We discuss the more interesting transition rules. Rules (Lts Com), (Lts !Com) and (Lts Run) closely mimic the corresponding reduction rules. These transitions do not require interaction with the external environment, so the label $l{\cdot}\tau$ requires only the existence of location l. Rule (Lts In) provides a first-order output message from the environment which can be used to analyze the continuation of an input process by deriving a further transition using the communication rules[1]. Rule (Lts Out) states that a potentially higher-order output

[1]Considering first order messages is enough because, since bisimilarity will be close with respect to parallel composition, the effect of higher order messages will be simulated by trigger definitions.

Figure 5.6: Labelled transition system

(Lts Com)
$$\overline{l{\cdot}c}\langle\widetilde{\pi}\sigma\rangle \mid l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \xrightarrow{l\cdot\tau} \boldsymbol{P}\sigma$$

(Lts !Com)
$$\overline{l{\cdot}c}\langle\widetilde{\pi}\sigma\rangle \mid !l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \xrightarrow{l\cdot\tau} \boldsymbol{P}\sigma \mid !l{\cdot}c(\widetilde{\pi}).\boldsymbol{P}$$

(Lts Run)
$$(x,\widetilde{\pi})\boldsymbol{P} \circ \langle l,\widetilde{\pi}\sigma\rangle \xrightarrow{l\cdot\tau} \boldsymbol{P}\{{}^{l}/_{x}\}\sigma$$

(Lts In)
$$\frac{\mathit{scripts}(\widetilde{v}) = \emptyset}{0 \xrightarrow{l\cdot c(\widetilde{v})} \overline{l{\cdot}c}\langle\widetilde{v}\rangle}$$

(Lts Out)
$$\frac{\mathfrak{X}(\widetilde{v}) = (\widetilde{v'};\Theta^{\widetilde{k}})}{\overline{l{\cdot}c}\langle\widetilde{v}\rangle \xrightarrow{(\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v'}\rangle} \Theta^{\widetilde{k}}}$$

(Lts Trigger)
$$\frac{\mathfrak{X}(\widetilde{v}) = (\widetilde{v'};\Theta^{\widetilde{k}}) \quad j \notin \{\widetilde{k}\}}{j \circ \langle l,\widetilde{v}\rangle \xrightarrow{(\widetilde{k})\overline{l{\cdot}j}\langle\widetilde{v'}\rangle} \Theta^{\widetilde{k}}}$$

(Lts Open)
$$\frac{K \xrightarrow{(\widetilde{a},\widetilde{k})\overline{l{\cdot}t}\langle\widetilde{v}\rangle} K' \quad a \in \mathit{fn}(\widetilde{v}) \setminus \{\widetilde{a},t\}}{(\nu\, a)K \xrightarrow{(a,\widetilde{a},\widetilde{k})\overline{l{\cdot}t}\langle\widetilde{v}\rangle} K'}$$

(Lts Req)
$$\frac{\mathfrak{X}_{\mathcal{Q}}(p) = (p',\Theta^{\widetilde{k}}) \quad \mathit{scripts}(T) = \emptyset \quad T = \mathtt{r}[\,U_1\,]_{\mathsf{l}} \ldots \mathtt{r}[\,U_n\,]_{\mathsf{l}}\varnothing}{l{\cdot}\mathsf{req}_p\langle c\rangle \xrightarrow{(\widetilde{k})l\cdot\mathsf{req}\langle p'\rangle(T)} \overline{l{\cdot}c}\langle T\rangle \mid \Theta^{\widetilde{k}}}$$

(Lts Def)
$$\frac{\mathit{scripts}(\sigma) = \emptyset}{\langle k \Leftarrow (x,\widetilde{\pi})\boldsymbol{P}\rangle \xrightarrow{l\cdot k(\widetilde{\pi}\sigma)} \langle k \Leftarrow (x,\widetilde{\pi})\boldsymbol{P}\rangle \mid \boldsymbol{P}\{{}^{l}/_{x}\}\sigma}$$

(Lts Go)
$$\overline{l{\cdot}\mathsf{go}\, m.P \xrightarrow{m\cdot\tau} P}$$

(Lts Res)
$$\frac{K \xrightarrow{\alpha_l} K' \quad a \notin n(\alpha_l)}{(\nu\, a)K \xrightarrow{\alpha_l} (\nu\, a)K'}$$

(Lts Par)
$$\frac{K \xrightarrow{\alpha_l} K' \quad \mathit{rel}(\alpha_l, L)}{K \mid L \xrightarrow{\alpha_l} K' \mid L}$$

(Lts Struct)
$$\frac{K \equiv L \xrightarrow{\alpha_l} L' \equiv K'}{K \xrightarrow{\alpha_l} K'}$$

Notation: $\mathit{rel}(\alpha_l, K) \stackrel{\text{def}}{=} \mathit{bn}(\alpha_l) \cap \mathit{fn}(K) = \emptyset$.
Convention: in this table c ranges over $\mathcal{C}_{\mathrm{p}} \cup \mathcal{C}_{\mathrm{s}}$, and t ranges over $\mathcal{C}_{\mathrm{p}} \cup \mathcal{C}_{\mathrm{s}} \cup \mathcal{Y}$.

$\overline{l{\cdot}c}\langle\widetilde{v}\rangle$ evolves to the definitions that are extracted from $\widetilde{v}$ to obtain $\widetilde{v'}$, and carries in the label the first-order version of the process. The intuition is that a bisimilar process will be required to perform the same first-order transition, and a potential incompatibility between the original higher-order messages will be detected by analyzing the resulting definitions $\Theta^{\widetilde{k}}$. Rule (Lts Trigger) states that the application of a trigger name to the potentially higher-order parameters $\widetilde{v}$ evolves to the definitions that are extracted from $\widetilde{v}$ to obtain $\widetilde{v'}$, and carries in the label the first-order version of the process, similarly to the case for output. Rule (Lts Open) is standard. Not that it applies to transitions originated using (Lts Out) or (Lts Trigger). Rule (Lts Req) can be interpreted as the combination of an output of p and an input of T on a special name req. Rule (Lts Def) analyzes the script of a definition for all its possible

(first-order) input parameters[2].

The sample query and update language. We conclude this section by extending Sam (Definition 2.3.1) to deal with trigger names, and showing that it respects our assumptions (Observation 5.1.5).

Definition 5.1.4 (Sam$^{\#}$) *The query language* Sam$^{\#}$ *is defined as* Sam*, with the exception that trees can contain also trigger names in the same position as scripts (i.e. within* $\langle - \rangle$*). The extraction relation* $\mathfrak{X}_{\mathcal{Q}}$*, and the functions* scripts *and* triggers *are defined on* Sam$^{\#}$ *by*

$$\frac{\mathfrak{X}(\boldsymbol{V}) = (\boldsymbol{U}; \Theta)}{\mathfrak{X}_{\mathcal{Q}}(\widehat{p}(\pi)\boldsymbol{V}) = (\widehat{p}(\pi)\boldsymbol{U}; \Theta)}$$

$$\mathit{scripts}(\widehat{p}(\pi)\boldsymbol{V}) = \mathit{scripts}(\boldsymbol{V}) \qquad \mathit{triggers}(\widehat{p}(\pi)\boldsymbol{V}) = \mathit{triggers}(\boldsymbol{V})$$

Observation 5.1.5 (Properties of Sam$^{\#}$) *(i)* Sam$^{\#}$ *is script independent, and (ii)* $\mathfrak{X}_{\mathcal{Q}}$ *for* Sam$^{\#}$ *respects Definition 5.1.2.*

Proof. Point (i) follows by induction on the derivation of $\mathfrak{E}$. The idea is that query evaluation does not depend on the structure of scripts, and by rule (Eval Match) only scripts coming from the query and the input tree can occur in the result and the output tree. Point (ii) follows by induction on the derivation of $\mathfrak{X}_{\mathcal{Q}}$. □

5.2 Domain bisimilarity

We introduce our bisimulation equivalence. The intuition is that, when two bisimilar processes are running in a location domain Λ, if a process makes an action α_l with $l \in \Lambda$ then the other one must be able to mimic it, possibly relying on the existence of other locations in Λ.

Since the location domain can be extended to $\Lambda \cup \Lambda'$ by composing networks, we need to make sure that also the actions mentioning locations in Λ' are matched, this time within a larger relation parameterized by $\Lambda \cup \Lambda'$. [3]

The definition of bisimilarity relies on the following derived transition relations.

Definition 5.2.1 (Derived Transition Relations) *Consider the lts defined in Figure 5.6. Given* $l \in \Lambda$*, we use the notation*

$$\xrightarrow{\tau}_{\Lambda} \stackrel{\text{def}}{=} \xrightarrow{l\cdot\tau}; \quad \xrightarrow{l\cdot\tau}\!\!\!\!\rightarrow_{\Lambda} \stackrel{\text{def}}{=} \xrightarrow{\tau *}_{\Lambda}; \quad \xrightarrow{\alpha_l}\!\!\!\!\rightarrow_{\Lambda} \stackrel{\text{def}}{=} \xrightarrow{\tau *}_{\Lambda} \circ \xrightarrow{\alpha_l} \circ \xrightarrow{\tau *}_{\Lambda} \quad \textit{when } \alpha_l \neq l\cdot\tau.$$

[2] Both in (Lts Def) and (Lts In) we do not need to consider higher-order values. This is due to the fact that the bisimilarity relation that we will consider turns out to be closed with respect to parallel composition with definitions (Theorem 5.3.15), and hence already takes into account the effects of scripts received from the environment.

[3] Our domain bisimilarity should not be confused with the notion of *translocating equivalence* of the Nomadic Pict language [77]. In Nomadic Pict, the set of agent names considered in a bisimulation proof can grow dynamically, and bisimulation must be explicitly closed under functions assigning agents to locations (drawn from a fixed set). This is needed in order to ensure the closure of bisimilarity under parallel contexts. In X$d\pi$ instead, it is the set of locations considered in a bisimulation proof that can grow dynamically, and this feature is used to reason about which locations can be relied upon, and which may be prone to failures (see Section 4.4).

Definition 5.2.2 (Domain Bisimilarity) *A family of symmetric relations on configurations (indexed with sets of locations)* $\dot{\approx} = \{\dot{\approx}_\Lambda \ : \ \Lambda \subseteq \mathcal{L}\}$ *is a* domain bisimulation *if* $K \dot{\approx}_\Lambda L$ *and* $K \xrightarrow{\alpha_l} K'$ *implies:*

1. *if* $l \in \Lambda$ *with* $rel(\alpha_l, L)$ *then* $L \stackrel{\alpha_l}{\twoheadrightarrow}_\Lambda L'$ *and* $K' \dot{\approx}_\Lambda L'$*;*
2. *if* $l \notin \Lambda$ *then* $K \dot{\approx}_{\Lambda \cup \{l\}} L$*.*

Domain bisimilarity $\approx = \{\approx_\Lambda \ : \ \Lambda \subseteq \mathcal{L}\}$ *is the (point-wise) largest domain bisimulation: if* $\dot{\approx}$ *is a domain bisimulation, then* $\dot{\approx}_\Lambda \subseteq \approx_\Lambda$ *for all* Λ*. Two open processes* $\boldsymbol{P}, \boldsymbol{Q}$ *are* Λ*-bisimilar if and only if for all closing substitutions* σ*,* $\boldsymbol{P}\sigma \approx_\Lambda \boldsymbol{Q}\sigma$*.*

Remark 5.2.3 (Initial Elements) *To show* $K \approx_\Lambda L$ *for a specific* Λ*, we can exhibit a domain bisimulation* $\dot{\approx} = \{\dot{\approx}_\Delta \ : \ \Delta \subseteq \mathcal{L}\}$ *such that* $K \dot{\approx}_\Lambda L$ *and* $\dot{\approx}_\Delta$ *is the empty set for all* Δ *smaller than* Λ*.*

Since our definition of domain bisimulation is non-standard, we need to argue that the largest domain bisimulation exists. In Section 5.3.4, we shall prove that it is indeed the case using a fixed-point characterization. Domain bisimilarity is a coinductive relation, preserved by structural congruence and monotonic in the domain Λ (Section 5.3.1). Under the mild assumption that the query language does not depend on scripts (Definition 5.1.1), domain bisimilarity enjoys two important properties which make it a useful proof method for domain congruence:

- domain bisimilarity is a *congruence*, that is embedding open processes in full contexts preserves bisimilarity (Section 5.3.2);
- domain bisimilarity is a *sound approximation* of the domain congruence induced by request observables, that is if two processes are bisimilar then they are request-congruent (Section 5.3.3).

We now give a first example of the proof method. Larger examples are given in Chapter 6.

Example 5.2.4 (Proof Method) *Recall the asynchrony law of Remark 4.4.4. It states that a communication buffer cannot be distinguished from the empty process. By definition,* $!\boldsymbol{l}\cdot\mathsf{FW}(\boldsymbol{a}, \boldsymbol{a}, \widetilde{\pi}) \approx_\Lambda \mathbf{0}$ *if for any closing substitution* σ*,* $(!\boldsymbol{l}\cdot\mathsf{FW}(\boldsymbol{a}, \boldsymbol{a}, \widetilde{\pi}))\sigma \approx_\Lambda \mathbf{0}\sigma$*. Given an arbitrary* σ *we have that* $(!\boldsymbol{l}\cdot\mathsf{FW}(\boldsymbol{a}, \boldsymbol{a}, \widetilde{\pi}))\sigma = !l\cdot\mathsf{FW}(a, a, \widetilde{\pi})$ *for some* a, l*. To show that* $!l\cdot\mathsf{FW}(a, a, \widetilde{\pi}) \approx_\Lambda \mathbf{0}$*, we need to give a domain bisimulation* $\dot{\approx} = \{\dot{\approx}_\Delta\}$ *such that* $\dot{\approx}_\Lambda$ *contains the two processes. Since structural congruence preserves bisimilarity (Proposition 5.3.1), we reason up-to* $\equiv$*.*

For each Δ*, we begin with a relation* $\mathcal{R}^0_\Delta = \{(!l\cdot\mathsf{FW}(a, a, \widetilde{\pi}), \mathbf{0})\}$ *containing the pair that we want to prove bisimilar. By definition of bisimilarity, we must close the relation under transitions. Due to* (Lts In) *we must close the relation under parallel compositions with arbitrary output processes:* $\mathcal{R}^1_\Delta = \{(M \mid !l\cdot\mathsf{FW}(a, a, \widetilde{\pi}), M)\}$ *where* $M = \prod_{0 \le i \le n} \overline{l_i \cdot c_i}\langle \widetilde{v_i} \rangle$*,* $\mathit{dom}(M) \subseteq \Delta$ *and* $\mathit{scripts}(\widetilde{v_i}) = \emptyset$ *(note that* $\mathcal{R}^1_\Delta = \mathcal{R}^0_\Delta$ *if* $n = 0$*).*
The possible tau transitions arising from the interaction of $!l\cdot\mathsf{FW}(a, a, \widetilde{\pi})$ *and an*

output $\overline{l{\cdot}a}\langle\widetilde{v}\rangle$ *where* $\widetilde{v} = \widetilde{\pi}\sigma$ *are already covered because by* (Lts !Com) *and* (Lts Struct), $\overline{l{\cdot}a}\langle\widetilde{v}\rangle \mid !l{\cdot}\mathsf{FW}(a, a, \widetilde{\pi}) \xrightarrow{l\cdot\tau} \overline{l{\cdot}a}\langle\widetilde{v}\rangle \mid !l{\cdot}\mathsf{FW}(a, a, \widetilde{\pi})$. *Again by definition of bisimilarity, we must make the relation symmetric, hence we conclude with* $\dot{\approx}_\Delta = \mathcal{R}^1_\Delta \cup (\mathcal{R}^1_\Delta)^{-1}$.

Incompleteness. In general, domain bisimilarity is a more restrictive equivalence than request congruence. The property is intrinsic to our choice of giving a proof method parametric in the chosen query and update language. In fact, without specializing the labelled transition system to a particular language, we are forced to distinguish request transitions as soon as queries are syntactically different. On the other hand, equivalences dependent on specific knowledge of the semantics of queries would lead to optimizations which are no longer correct when the query language changes.

Example 5.2.5 (Incompleteness) *Consider the query language* $\mathsf{Sam}^\#$ *and the process definition*

$$X(\mathtt{a},\mathtt{b}) \stackrel{\text{def}}{=} (\nu\, c)(\overline{l{\cdot}c} \mid !l{\cdot}c.(\nu\, e)\begin{pmatrix} l{\cdot}\mathsf{req}_{(x)\mathtt{a}}\langle e\rangle \mid \\ l{\cdot}e(x).(\nu\, e')(l{\cdot}\mathsf{req}_{(x)\mathtt{b}}\langle e'\rangle \mid l{\cdot}e'(x).\overline{l{\cdot}c}) \end{pmatrix})$$

The process loops, replacing at each iteration whatever tree is at l *first with* $\mathtt{a}$ *and then with* $\mathtt{b}$. *We have* $X(\mathtt{a},\mathtt{b}) \sim_r^{\{l\}} X(\mathtt{b},\mathtt{a})$, *because once the two processes are inserted in the same store, they can always reduce to each other. On the other hand, we have that* $X(\mathtt{a},\mathtt{b}) \not\approx_{\{l\}} X(\mathtt{b},\mathtt{a})$ *because the request transitions cannot be matched.*

5.3 Results and proofs

This section describes the properties of domain bisimilarity and gives the formal proofs. It is interesting for the reader wishing to see the non-standard technical details, but it is not necessary for understanding the rest of the thesis.

To follow more easily certain common steps in the proofs, it may be helpful to keep in mind that: private and service channel names are distinct; a script is well-formed only if it has no free private channel or trigger names; configurations are well-formed if for any trigger name there is at most one definition.

5.3.1 Basic properties

We study some basic properties of domain bisimilarity which will be useful to prove the main results of congruence and soundness. A first property is that structural congruence preserves bisimilarity. We will use this implicitly in the rest of the chapter.

Proposition 5.3.1 (Bisimilarity Up-To Structural Congruence) *If* $K \approx_\Lambda L$, $K \equiv K'$ *and* $L \equiv L'$, *then* $K' \approx_\Lambda L'$.

Proof. The family of relations $\dot{\approx}$ with generic element

$$\dot{\approx}_\Delta = \left\{(K', L') \;:\; K' \equiv K \approx_\Delta L \equiv L'\right\}$$

is a domain bisimulation. Follows by using rule (Lts Struct). □

By definition of bisimilarity, the smaller the domain Λ, the less likely that two processes are bisimilar. In fact, we need to check for matching actions first in Λ, then in any Λ' containing Λ. The underlying intuition is that if we can rely on a larger set of locations to be connected to the network, then we can perform more optimizations.

Proposition 5.3.2 (Monotonicity) *Domain bisimilarity is monotonic: for all sets of locations Λ, Λ', if $\Lambda \subsetneq \Lambda'$ then $\approx_\Lambda \subsetneq \approx_{\Lambda'}$.*

Proof.

($\subseteq$) Follows by Definition 5.2.2, noticing that using rule (Lts In) it is always possible to make an input action at location l, for any l not in Λ.

($\subsetneq$) If $\Lambda \subsetneq \Lambda'$ then there exists an m such that $m \in \Lambda' \setminus \Lambda$. Consider the two processes $P = \overline{l{\cdot}a}$ and $Q = l{\cdot}\mathsf{go}\, m.m{\cdot}\overline{l{\cdot}a}$, where $l \neq m$. Clearly, $P \not\approx_\Lambda Q$ because $P \xrightarrow{\overline{l{\cdot}a}} \mathbf{0}$ but there is no Q' such that $Q \overset{\overline{l{\cdot}a}}{\twoheadrightarrow_\Lambda} Q'$. To show that $P \approx_{\Lambda'} Q$, let $\mathcal{R}_\Delta$ be the set containing the pairs

$$(M \mid P, M \mid Q), \quad (M \mid P, M \mid m{\cdot}\overline{l{\cdot}a}), \quad (M, M)$$

for any M of the form

$$\prod_{0 \le i \le n} \overline{l_i{\cdot}c_i}\langle \widetilde{v_i} \rangle,\ \mathit{dom}(M) \subseteq \Delta$$

where $\mathit{scripts}(\widetilde{v_i}) = \emptyset$ for all i. The family $\dot{\approx}$, where $\dot{\approx}_\Delta = \mathcal{R}_\Delta \cup (\mathcal{R}_\Delta)^{-1}$ for each Δ containing Λ' and $\dot{\approx}_\Delta = \emptyset$ otherwise, is a domain bisimulation containing (P, Q), hence $P \approx_{\Lambda'} Q$.

□

The lemma given below is a standard technical lemma relating the transitions in the lts with the syntactic structure of configurations, up-to structural congruence. It is used in many proofs, sometimes implicitly.

Lemma 5.3.3 (Transition Correspondence) *The transitions of the lts are in close correspondence with the structure of configurations.*

1. $K \xrightarrow{(\widetilde{a},\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v}\rangle} K'$ *if and only if* $K \equiv (\nu\, \widetilde{a})(L \mid \overline{l{\cdot}c}\langle\widetilde{v'}\rangle)$ *where* $c \notin \{\widetilde{a}\}$, $\{\widetilde{a}\} \subseteq \mathit{fn}(\widetilde{v'})$ *and* $K' \equiv L \mid \Theta^{\widetilde{k}}$ *where* $\mathfrak{X}(\widetilde{v'}) = (\widetilde{v}; \Theta^{\widetilde{k}})$.

2. $K \xrightarrow{l{\cdot}c(\widetilde{v})} K'$ *if and only if* $K' \equiv K \mid \overline{l{\cdot}c}\langle\widetilde{v}\rangle$, $\mathit{scripts}(\widetilde{v}) = \emptyset$ *and* $\mathit{rel}(l{\cdot}c(\widetilde{v}), K)$.

3. $K \xrightarrow{l{\cdot}\tau} K'$ *if and only if*

- $K \equiv (\nu\, \widetilde{a})(L \mid l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \mid \overline{l{\cdot}c}\langle\widetilde{\pi}\sigma\rangle)$ *and* $K' \equiv (\nu\, \widetilde{a})(L \mid \boldsymbol{P}\sigma)$, *or*
- $K \equiv (\nu\, \widetilde{a})(L \mid !l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \mid \overline{l{\cdot}c}\langle\widetilde{\pi}\sigma\rangle)$ *and* $K' \equiv (\nu\, \widetilde{a})(L \mid !l{\cdot}c(\widetilde{\pi}).\boldsymbol{P} \mid \boldsymbol{P}\sigma)$, *or*
- $K \equiv (\nu\, \widetilde{a})(L \mid (x, \widetilde{\pi})\boldsymbol{P} \circ \langle l, \widetilde{\pi}\sigma\rangle)$ *and* $K' \equiv (\nu\, \widetilde{a})(L \mid \boldsymbol{P}\{{}^{l}/_{x}\}\sigma)$, *or*
- $K \equiv (\nu\, \widetilde{a})(L \mid m{\cdot}\mathsf{go}\ \ l.P)$ *and* $K' \equiv (\nu\, \widetilde{a})(L \mid P)$.

4. $K \xrightarrow{(\widetilde{k})l{\cdot}\mathsf{req}\langle p\rangle(T)} K'$ *if and only if* $K \equiv (\nu\, a)(L \mid l{\cdot}\mathsf{req}_{p'}\langle c\rangle)$ *and* $K' \equiv (\nu\, a)(L \mid \overline{l{\cdot}c}\langle T\rangle \mid \Theta^{\widetilde{k}})$ *for some* p' *such that* $\mathfrak{X}_{\mathcal{Q}}(p') = (p, \Theta^{\widetilde{k}})$ *and some* T *such that* $\mathit{scripts}(T) = \emptyset$ *and* T *has the form* $\mathtt{r}[\,U_1\,]_{\mathtt{l}} \ldots \mathtt{r}[\,U_n\,]_{\mathtt{l}}\varnothing$.

5. $K \xrightarrow{l{\cdot}k(\widetilde{\pi}\sigma)} K'$ *if and only if* $K \equiv L \mid \langle k \Leftarrow (x, \widetilde{\pi})\boldsymbol{P}\rangle$ *and* $K' \equiv K \mid \langle k \Leftarrow (x, \widetilde{\pi})\boldsymbol{P}\rangle \mid \boldsymbol{P}\{{}^{l}/_{x}\}\sigma$ *and* $\mathit{scripts}(\sigma) = \emptyset$.

6. $K \xrightarrow{(\widetilde{a},\widetilde{k})\overline{l{\cdot}j}\langle\widetilde{v}\rangle} K'$ *if and only if* $K \equiv (\nu\, \widetilde{a})(L \mid j \circ \langle l, \widetilde{v'}\rangle)$ *where* $j \notin \{\widetilde{k}\}$, $\{\widetilde{a}\} \subseteq \mathit{fn}(\widetilde{v'})$ *and* $K' \equiv L \mid \Theta^{\widetilde{k}}$ *where* $\mathfrak{X}(\widetilde{v'}) = (\widetilde{v}; \Theta^{\widetilde{k}})$.

Proof.

($\Longleftarrow$) Follows easily by definition of lts.

($\Longrightarrow$) By induction on the depth n of the derivation tree in the premise for the labelled transition. We give the case for bound output as an example (point 1).

($n = 0$) Suppose $K \xrightarrow{(\widetilde{a},\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v}\rangle} K'$ is derived by directly applying (Lts Out). It must be the case that $K = \overline{l{\cdot}c}\langle\widetilde{v'}\rangle$, where $\mathfrak{X}(\widetilde{v'}) = (\widetilde{v}; \Theta^{\widetilde{k}})$ and $K' = \Theta^{\widetilde{k}}$.

($n = m + 1$) A derivation of depth $m{+}1$ for $K \xrightarrow{(\widetilde{a},\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v}\rangle} K'$ must be obtained by applying one of the rules (Lts Res), (Lts Par), (Lts Struct) or (Lts Open) to a derivation of depth m. The case for (Struct) is trivial. If rule (Res) is applied then we must have that $(\nu\, d)K_1 \xrightarrow{(\widetilde{a},\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v}\rangle} (\nu\, d)K_1'$, where $d \notin \mathit{n}((\widetilde{a},\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v}\rangle)$, follows from the premise $K_1 \xrightarrow{(\widetilde{a},\widetilde{k})\overline{l{\cdot}c}\langle\widetilde{v}\rangle} K_1'$. By inductive hypothesis, $K_1 \equiv (\nu\, \widetilde{a})(L \mid \overline{l{\cdot}c}\langle\widetilde{v'}\rangle)$ where $c \notin \{\widetilde{a}\}$, $\{\widetilde{a}\} \subseteq \mathit{fn}(\widetilde{v'})$ and $K_1' \equiv L \mid \Theta^{\widetilde{k}}$ where $\mathfrak{X}(\widetilde{v'}) = (\widetilde{v}; \Theta^{\widetilde{k}})$. Since $\mathfrak{X}$ does not affect session channels, $d \notin \mathit{fn}(\widetilde{v'}) \setminus \{\widetilde{a}\}$. By structural congruence, $(\nu\, d)K_1 \equiv (\nu\, \widetilde{a})((\nu\, d)(L) \mid \overline{l{\cdot}c}\langle\widetilde{v'}\rangle)$, and $(\nu\, d)K_1' \equiv (\nu\, d)(L) \mid \Theta^{\widetilde{k}}$. The cases for (Par) and (Open) are similar.

□

The next step towards the main proofs consists of generalizing the variant lemma of [44] to bijective substitutions (here called *switchings*) on channel and trigger names[4]. By using switchings rather than generic substitutions we obtain a purely

[4]Our approach is reminiscent of the permutation-based approach to abstract syntax developed by Gabbay and Pitts [31]. In particular, it may be interesting in future work to compare our use of switchings with the work of Gabbay [30] on the π-calculus.

coinductive proof. Below we let a, b, c range over channel or trigger names, and we consider only well-sorted substitutions (replacing channels for channels and triggers for triggers).

Definition 5.3.4 (Switching) *Given a term t with a function fn returning its free names, a* switching $^{a\frown b}$ *is a bijective substitution $\{^c/_a, {}^a/_b\}\{^b/_c\}$ such that $c \notin fn(t) \cup \{a, b\}$. We denote by $^{\widetilde{a}\frown\widetilde{b}}$ the switching $^{a_1\frown b_1} \ldots {}^{a_n\frown b_n}$ where both $\widetilde{a}$ and $\widetilde{b}$ are vectors of distinct names.*

Observation 5.3.5 (Switching Properties) *Switching is self-dual $(K^{a\frown b})^{a\frown b} = K$ and symmetric $K^{a\frown b} = K^{b\frown a}$.*

Proof. Both properties follow from the definition of switching and substitution. □

We note below that both the extraction and the transition relations do not depend on specific names, hence they are fully compatible with switching, and α-conversion.

Lemma 5.3.6 (Switching Extraction) *Extraction preserves switching: if $\mathfrak{X}(\widetilde{v}) = (\widetilde{v'}, \Theta^{\widetilde{k}})$ then $\mathfrak{X}(\widetilde{v}^{a\frown b}) = (\widetilde{v'}^{a\frown b}; (\Theta^{\widetilde{k}})^{a\frown b})$.*

Proof. By a simple induction on the derivation of $\mathfrak{X}(\widetilde{v}) = (\widetilde{v'}, \Theta^{\widetilde{k}})$, where the base case for queries uses Definition 5.1.2. □

Lemma 5.3.7 (Switching Transitions) *If $K \xrightarrow{\alpha_l} K'$ then $K^{a\frown b} \xrightarrow{\alpha_l^{a\frown b}} K'^{a\frown b}$, provided that $bn(\alpha_l) \cap fn(K^{a\frown b}) \cap \{a, b\} = \emptyset$.*

Proof. By case analysis on α_l, using Lemma 5.3.3. We show the case for the bound output when the name of the channel used for output occurs in the switching; the other cases are simpler. Let $\rho = ^{a\frown b}$. Without loss of generality, suppose that $K \xrightarrow{\alpha_l} K'$ where $\alpha_l = (b, \widetilde{b})\overline{l{\cdot}a}\langle\widetilde{v'}\rangle$ and b does not occur in $\widetilde{b}$. By Lemma 5.3.3, $K \equiv (\nu\, b, \widetilde{b})(L \mid \overline{l{\cdot}a}\langle\widetilde{v}\rangle)$ where $a \notin \left\{b, \widetilde{b}\right\}$, $\left\{b, \widetilde{b}\right\} \subseteq fn(\widetilde{v})$ and $K' \equiv L \mid \Theta^{\widetilde{k}}$ where $\mathfrak{X}(\widetilde{v}) = (\widetilde{v'}; \Theta^{\widetilde{k}})$. Let c be a fresh name. By α-conversion and Observation 5.3.5, we have $K \equiv (\nu\, c, \widetilde{b})(L\{^c/_b\} \mid \overline{l{\cdot}a}\langle\widetilde{v}\{^c/_b\}\rangle)\rho\rho$. Applying the inner switching, we obtain $K \equiv (\nu\, c, \widetilde{b})(L\{^c/_b\}\rho \mid \overline{l{\cdot}b}\langle\widetilde{v}\{^c/_b\}\rho\rangle)\rho$. Since $\{^c/_b\}$ has replaced b with c,

$$K \equiv (\nu\, c, \widetilde{b})(L\{^c/_b\}\{^b/_a\} \mid \overline{l{\cdot}b}\langle\widetilde{v}\{^c/_b\}\{^b/_a\}\rangle)\rho$$

Since a does not appear free anymore, we can alpha-convert c with a in the term above, obtaining

$$K \equiv (\nu\, a, \widetilde{b})(L\{^c/_b\}\{^b/_a\}\{^a/_c\} \mid \overline{l{\cdot}b}\langle\widetilde{v}\{^c/_b\}\{^b/_a\}\{^a/_c\}\rangle)\rho$$

By definition of switching, $K \equiv (\nu\, a, \widetilde{b})(L\rho \mid \overline{l{\cdot}b}\langle\widetilde{v}\rho\rangle)\rho$. By Observation 5.3.5, $K\rho \equiv (\nu\, a, \widetilde{b})(L\rho \mid \overline{l{\cdot}b}\langle\widetilde{v}\rangle\rho)$. By Lemma 5.3.6, $\mathfrak{X}(\widetilde{v}\rho) = (\widetilde{v'}\rho; \Theta^{\widetilde{k}}\rho)$. By (Lts Out), $K\rho \xrightarrow{\alpha_l\rho} K'\rho$. □

The lemma below shows that bisimilarity is closed with respect to switchings, a property needed to show that it is transitive.

Lemma 5.3.8 (Variant) *(i) If $K \approx_\Lambda L$ then $K^{a\frown b} \approx_\Lambda L^{a\frown b}$. (ii) If $b \notin \mathit{fn}(K, L)$ then $K \approx_\Lambda L \Longrightarrow K\{{}^b/_a\} \approx_\Lambda L\{{}^b/_a\}$.*

Proof. (i) Let $\rho = {}^{a\frown b}$. We show that the family $\dot{\approx}$ with generic element $\dot{\approx}_\Delta = \{(K\rho, L\rho) \ : \ K \approx_\Delta L\}$ is a domain bisimulation. Assume $K \approx_\Delta L$ for some Δ. Suppose $K\rho \xrightarrow{\alpha_l} K'$ and $l \notin \Delta$. By $K \approx_\Delta L$, $K \approx_{\Delta\cup\{l\}} L$, hence $(K\rho, L\rho) \in \dot{\approx}_{\Delta\cup\{l\}}$. Suppose instead $l \in \Delta$ and $\mathit{rel}(\alpha_l, L\rho)$. By Lemma 5.3.7, $K \xrightarrow{\alpha_l\rho} K'\rho$ with $\mathit{rel}(\alpha_l\rho, L)$. By bisimilarity, $L \xrightarrow{\alpha_l\rho}\!\!\!\!\rightarrow_\Delta L'$ with $K'\rho \approx_\Delta L'$. By Lemma 5.3.7, $L\rho \xrightarrow{\alpha_l}\!\!\!\!\rightarrow_\Delta L'\rho$. By definition, $(K'\rho\rho, L'\rho) \in \dot{\approx}_\Delta$. We conclude because, by Observation 5.3.5, $K'\rho\rho = K'$. (ii) Follows from (i) by definition of switching. □

Domain bisimilarity is an equivalence relation. This property is very important, because in the rest of the proofs in this chapter we will often rely on symmetry and transitivity.

Proposition 5.3.9 (Equivalence) *Domain bisimilarity is an equivalence relation.*

Proof. Reflexivity and symmetry are immediate. Transitivity states that if $K \approx_\Lambda M$ and $M \approx_\Lambda L$ then $K \approx_\Lambda L$. We show that the family $\dot{\approx}$ with generic element

$$\dot{\approx}_\Delta = \{(K, L) \ : \ K \approx_\Delta M,\ M \approx_\Delta L\}$$

is a domain bisimulation. Let Δ be arbitrary and suppose $K \xrightarrow{\alpha_l} K'$ with $l \notin \Delta$. By Definition 5.2.2, $K \approx_{\Delta\cup\{l\}} M$ and $M \approx_{\Delta\cup\{l\}} L$, hence $(K, L) \in \dot{\approx}_{\Delta\cup\{l\}}$. If $l \in \Delta$ and $\mathit{rel}(\alpha_l, L)$ then there are two cases, determined by the relevance of α_l to M. If $\mathit{rel}(\alpha_l, M)$, the proof is straightforward. If α_l is not relevant to M, the action α_l must necessarily have some bound names $\widetilde{c}$ such that $\{\widetilde{c}\} \subseteq \mathit{fn}(M)$. By the second premise of (Lts Par) used to derive the bound transition, $\{\widetilde{c}\} \cap \mathit{fn}(K) = \emptyset$. Let $\widetilde{a}$ have the same length as $\widetilde{c}$, and be such that $\{\widetilde{a}\} \cap \mathit{fn}(K, L, M) = \emptyset$. By Lemma 5.3.8, $K = K\{{}^{\widetilde{a}}/_{\widetilde{c}}\} \approx_\Delta M\{{}^{\widetilde{a}}/_{\widetilde{c}}\} = M'$. By the same argument, $M' \approx_\Delta L$. Since now $\mathit{rel}(\alpha_l, M')$, the proof is straightforward. □

5.3.2 Congruence

Our next objective is to show that domain bisimilarity is a congruence. We already know that it is an equivalence relation (Proposition 5.3.9), and that it preserves switchings (Lemma 5.3.8). We also know how to relate labelled transitions with the syntactic structure of configurations (Lemma 5.3.3). Using these tools, it is pretty easy to show that bisimilarity is closed under the restriction operator and, for processes, under prefixes. Most of the work in this section is dedicated to show *directly* closure under parallel composition. In contrast, Jeffrey and Rathke [46] for example show the soundness of their bisimilarity with respect to barbed congruence by using an auxiliary reduction-closed relation, which is closed under parallel composition. Showing the corresponding completeness result, they derive that also bisimilarity is closed under parallel composition. We cannot use their approach due to the inherent incompleteness of domain bisimilarity (see Section 5.2). Instead, we give a direct proof

Figure 5.7: Merge operator for configurations

$$\langle\!\langle \boldsymbol{P} \rangle\!\rangle = \boldsymbol{P} \qquad \text{(Merge Proc)}$$

$$\langle\!\langle (\nu\, c)K \rangle\!\rangle = (\nu c)\langle\!\langle K \rangle\!\rangle \qquad \text{(Merge Res)}$$

$$\langle\!\langle K \mid \langle k \Leftarrow A \rangle \rangle\!\rangle = \langle\!\langle K^{\langle k \Leftarrow A \rangle} \rangle\!\rangle \qquad \text{(Merge Def)}$$

of closure under parallel composition. This is possible in our framework because, exploiting the fact that scripts do not contain triggers or private names, we can use $\equiv$ to re-factor each configuration into a process part and a definition part in Lemma 5.3.14.

Lemma 5.3.10 (Restriction) *Bisimilarity is closed under restriction:* $K \approx_\Lambda L \Longrightarrow (\nu\, \widetilde{c})K \approx_\Lambda (\nu\, \widetilde{c})L$.

Proof. The family $\dot{\approx}$ with generic element

$$\dot{\approx}_\Delta = \{(K_1, L_1) \;:\; K_1 \equiv (\nu\, \widetilde{c})K,\, L_1 \equiv (\nu\, \widetilde{c})L,\, K \approx_\Delta L\}$$

is a domain bisimulation. Suppose $K_1 \xrightarrow{\alpha_l} K_1'$. The proof is by cases on α_l using Lemma 5.3.3. We show the case for $\alpha_l = l{\cdot}c(\widetilde{v})$ which is the most interesting. If $l \notin \Delta$, the proof is easy. Suppose $l \in \Delta$. By Lemma 5.3.3, $K_1' \equiv K_1 \,|\, \overline{l{\cdot}c}\langle\widetilde{v}\rangle$. By hypothesis, $K_1 \equiv (\nu\, \widetilde{c})K$, $K \approx_\Delta L$ and $L_1 \equiv (\nu\, \widetilde{c})L$. By α-conversion, $K_1 \equiv (\nu\, \widetilde{c'})K\{\widetilde{c'}/\widetilde{c}\}$ for a fresh tuple of names $\widetilde{c'}$. By (Lts Struct), (Lts Par) and (Lts In), $(\nu\, \widetilde{c'})K\{\widetilde{c'}/\widetilde{c}\} \xrightarrow{l{\cdot}c(\widetilde{v})} (\nu\, \widetilde{c'})(K\{\widetilde{c'}/\widetilde{c}\}) \,|\, \overline{l{\cdot}c}\langle\widetilde{v}\rangle$ and $K\{\widetilde{c'}/\widetilde{c}\} \xrightarrow{l{\cdot}c(\widetilde{v})} K\{\widetilde{c'}/\widetilde{c}\} \,|\, \overline{l{\cdot}c}\langle\widetilde{v}\rangle$. By Lemma 5.3.8, $K\{\widetilde{c'}/\widetilde{c}\} \approx_\Delta L\{\widetilde{c'}/\widetilde{c}\}$, hence $L\{\widetilde{c'}/\widetilde{c}\} \xrightarrow{l{\cdot}c(\widetilde{v})}\!\!\!\!\rightarrow_\Delta L' \approx_\Delta K\{\widetilde{c'}/\widetilde{c}\} \,|\, \overline{l{\cdot}c}\langle\widetilde{v}\rangle$. By (Lts Res) and freshness of $\widetilde{c'}$, $(\nu\, \widetilde{c'})(L\{\widetilde{c'}/\widetilde{c}\}) \xrightarrow{l{\cdot}c(\widetilde{v})}\!\!\!\!\rightarrow_\Delta (\nu\, \widetilde{c'})L'$.
By (Lts Struct), $(\nu\, \widetilde{c})L \xrightarrow{l{\cdot}c(\widetilde{v})}\!\!\!\!\rightarrow_\Delta L_1' \equiv (\nu\, \widetilde{c'})L'$. By α-conversion and freshness of $\widetilde{c'}$, $K_1' \equiv (\nu\, c')(K\{{}^{c'}/_{c}\} \,|\, \overline{l{\cdot}c}\langle\widetilde{v}\rangle)$, hence $K_1' \dot{\approx}_\Delta L_1'$. □

Following Jeffrey and Rathke [46], we define in Figure 5.7 a *merge* operator $\langle\!\langle - \rangle\!\rangle$ to reconstruct processes from configurations[5]. This operator plays a substantial rôle in showing that $\approx_\Lambda$ is closed under parallel composition.

Before showing the properties of the merge operator, we illustrate three simple properties of the extraction function: it does not remove trigger names; it associates a definition to each trigger name it introduces; and we can recover the initial term by substituting the new definitions in the result term.

Lemma 5.3.11 (Extraction Properties) *Suppose* $\mathfrak{X}(\widetilde{v}) = (\widetilde{v'}; \Theta^{\widetilde{k}})$. *The following properties hold:*

[5]The merge operator of [46] is partial due to a potential circularity of references between trigger names and definitions. Since scripted processes in definitions cannot contain triggers, our merge operator is total.

1. *if* $k \in \mathit{fn}(\widetilde{v})$ *then* $k \in \mathit{fn}(\widetilde{v'})$;
2. *if* $k \in \mathit{fn}(\widetilde{v'}) \setminus \{\widetilde{k}\}$ *then* $k \in \mathit{fn}(\widetilde{v})$;
3. $\widetilde{v} = \widetilde{v'}^{\Theta^{\tilde{k}}}$.

Proof. By induction on the structure of $\widetilde{v}$, using Definition 5.1.2. □

Since definitions can appear only at the top level and scripts cannot contain free private channel names, we can always use structural equivalence to factor any configuration into the parallel composition of a process and a group of definitions. We will make substantial use of this property to show closure under parallel composition of $\approx_\Lambda$. Merging a configuration corresponds to substituting the script in each definition for the corresponding trigger names in the process term of the configuration. Hence, the merge operator preserves the transitions that do not involve triggers names for which there is a corresponding definition. Moreover, if two configurations are bisimilar, then they must define the same trigger names.

Lemma 5.3.12 (Merge Properties) *The merge operator satisfies the following properties:*

1. *factorization: for any well-formed* K*, there exist a process* P *and a configuration* $\Theta^{\widetilde{k}}$ *such that* $K \equiv P \mid \Theta^{\widetilde{k}}$ *and* $\langle\!\langle K \rangle\!\rangle \equiv P^{\Theta^{\widetilde{k}}}$;
2. *transition preservation: for any* $\Theta^{\widetilde{k}}, \Theta^{\widetilde{j}}$*, if* $P \xrightarrow{\alpha_l} P'$ *and* $\left\{\widetilde{k}, \widetilde{j}\right\} \cap n(\alpha_l) = \emptyset$ *then* $\langle\!\langle P \mid \Theta^{\widetilde{k}} \rangle\!\rangle \mid \Theta^{\widetilde{j}} \xrightarrow{\alpha_l} \langle\!\langle P' \mid \Theta^{\widetilde{k}} \rangle\!\rangle \mid \Theta^{\widetilde{j}}$;
3. *If* $K \approx_\Lambda L$ *then* $K \equiv P \mid \Theta^{\widetilde{k}}$ *and* $L \equiv Q \mid \Omega^{\widetilde{k}}$*, and each pair of corresponding definitions contains scripts with the same patterns: that is,* $\langle k \Leftarrow (\widetilde{\pi})P_k \rangle$ *in* $\Theta^{\widetilde{k}}$ *implies* $\langle k \Leftarrow (\widetilde{\pi})Q_k \rangle$ *in* $\Omega^{\widetilde{k}}$*, and viceversa.*

Proof.

1. By induction on the structure of K.
2. By induction on the derivations of $\xrightarrow{\alpha_l}$, using Lemma 5.3.3.
3. Follows from point 1 above noticing that if one configuration contains a definition involving a trigger not present in the other configuration, or with different patterns, then it can do a labelled transition which cannot be matched.

□

The lemma below analyzes the relationship between bisimilarity and definitions. We start noting that if we remove from the two configurations the definitions for the same set of names, bisimilarity is preserved. Then, we note that the configurations obtained by duplicating existing definitions, using arbitrary fresh trigger names, remain bisimilar. These properties will be useful for showing that bisimilarity is closed under parallel composition.

Lemma 5.3.13 (Bisimilarity and Definitions) *Let K and L be well-formed configurations.*

1. *If $K \mid \Theta^{\widetilde{k}} \approx_\Lambda L \mid \Omega^{\widetilde{k}}$ then $K \approx_\Lambda L$.*
2. *If $K \mid \Theta^{\widetilde{k}} \approx_\Lambda L \mid \Omega^{\widetilde{k}}$ then $K \mid \Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k}\frown\widetilde{j}} \approx_\Lambda L \mid \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k}\frown\widetilde{j}}$.*

Proof.

1. The family $\dot{\approx}$ with generic element

$$\dot{\approx}_\Delta = \left\{ (K, L) \;\; : \;\; K \mid \Theta^{\widetilde{k}} \approx_\Delta L \mid \Omega^{\widetilde{k}} \right\}$$

   is a domain bisimulation. Follows by analyzing the transitions of K, using Lemma 5.3.8. The intuition is every transition by $K \mid \Theta^{\widetilde{k}}$ originating from K must be matched by $L \mid \Omega^{\widetilde{k}}$ using a (weak) transition originating from L alone, since $\Theta^{\widetilde{k}}$ and $\Omega^{\widetilde{k}}$ can perform only (Lts Def) transitions.

2. The family $\dot{\approx}$ with generic element

$$\dot{\approx}_\Delta = \left\{ (K \mid \Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k}\frown\widetilde{j}}, L \mid \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k}\frown\widetilde{j}}) \;\; : \;\; K \mid \Theta^{\widetilde{k}} \approx_\Delta L \mid \Omega^{\widetilde{k}} \right\}$$

   is a domain bisimulation. Follows by analysis of the transitions of $K \mid \Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k}\frown\widetilde{j}}$, by syntactic reasoning using Lemma 5.3.7. The intuition is that since $\Theta^{\widetilde{k}}$ and $\Theta^{\widetilde{k}\frown\widetilde{j}}$ have the same transitions up-to renaming of triggers, every process generated by trigger transitions from $\Theta^{\widetilde{k}\frown\widetilde{j}}$ could also be generated by $\Theta^{\widetilde{k}}$ alone, and all that is needed is to match the transition of $\Theta^{\widetilde{k}\frown\widetilde{j}}$ with a corresponding one by $\Omega^{\widetilde{k}\frown\widetilde{j}}$, which exists because $K \mid \Theta^{\widetilde{k}} \approx_\Lambda L \mid \Omega^{\widetilde{k}}$ and $\Omega^{\widetilde{k}}$ can match $\Theta^{\widetilde{k}}$.

□

Lemma 5.3.14 (Parallel Composition) *Bisimilarity is closed under parallel composition: $K \approx_\Lambda L \Longrightarrow K \mid M \approx_\Lambda L \mid M$.*

Proof. In order to show closure under parallel composition, we will identify a domain bisimulation $\dot{\approx}$ containing all the pairs of the form $(K \mid M, L \mid M)$ such that K is bisimilar to L, plus any other pair of terms generated by the labelled transition system. In particular, we must handle with care the terms generated by a communication steps between K (or L) and M involving scripts. The idea is that we represent explicitly, using the merge operators, the definitions corresponding to the communicated scripts. More in detail, by point 3 of Lemma 5.3.12 we know that, since $K \approx_\Lambda L$, then $K \equiv \Theta^{\widetilde{k}} \mid P'$ and $L \equiv \Omega^{\widetilde{k}} \mid Q'$ for some appropriate $P', \Theta^{\widetilde{k}}, Q', \Omega^{\widetilde{k}}$. Moreover, using point 1 of Lemma 5.3.12 we can rewrite $P' \equiv \langle\!\langle P \mid \Theta^{\widetilde{m}} \rangle\!\rangle$ and $Q' \equiv \langle\!\langle Q \mid \Omega^{\widetilde{m}} \rangle\!\rangle$ for some appropriate $P, \Theta^{\widetilde{m}}, Q, \Omega^{\widetilde{m}}$. That is, $K \equiv \Theta^{\widetilde{k}} \mid \langle\!\langle P \mid \Theta^{\widetilde{m}} \rangle\!\rangle$ and $L \equiv \Omega^{\widetilde{k}} \mid \langle\!\langle Q \mid \Omega^{\widetilde{m}} \rangle\!\rangle$. Again by point 1 of Lemma 5.3.12, we also know that $M \equiv \langle\!\langle \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \mid \Phi^{\widetilde{i}}$ for some

$R, \Phi^{\widetilde{i}}, \Phi^{\widetilde{n}}$. The terms $\Theta^{\widetilde{m}}, \Omega^{\widetilde{m}}$ and $\Phi^{\widetilde{n}}$ represent definitions corresponding to the scripts that respectively K, L or M may communicate in a future transition.

Our candidate bisimulation is the family $\dot{\approx}$ with generic element $\dot{\approx}_\Delta$ defined (up to $\equiv$) by the pairs

$$\big((\nu\,\widetilde{c})(\Theta^{\widetilde{k}} \mid \langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Theta^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \mid \Phi^{\widetilde{i}}), (\nu\,\widetilde{c})(\Omega^{\widetilde{k}} \mid \langle\!\langle Q \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Omega^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \mid \Phi^{\widetilde{i}})\big)$$

(where all the $\widetilde{h}, \widetilde{i}, \widetilde{j}, \widetilde{k}, \widetilde{m}, \widetilde{m}$ are distinct) such that

$$\Theta^{\widetilde{k}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j}} \mid P \approx_\Delta \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{j}} \mid Q$$

The extra terms $\Phi^{\widetilde{h}}, \Theta^{\widetilde{j}}, \Omega^{\widetilde{j}}$ represent the definitions corresponding to the scripts that respectively M, K or L may have communicated using a labelled transition to either K or L, or to M. Note that $\{(K \mid M, L \mid M) \;:\; K \approx_\Delta L\}$ is contained in $\dot{\approx}_\Delta$ (up to $\equiv$), by choosing $\Phi^{\widetilde{h}} = \Theta^{\widetilde{j}} = \Phi^{\widetilde{j}} = \mathbf{0}$ and $\widetilde{c}$ empty.

We now proceed to show that $\dot{\approx}$ is a domain bisimulation. For readability, we omit the subscript Δ on weak transitions $\stackrel{a}{\longrightarrow\!\!\!\!\!\rightarrow}_\Delta$, and we use the abbreviations

$$\begin{aligned}
K_* &= (\nu\,\widetilde{c})(\Theta^{\widetilde{k}} \mid \langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Theta^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \mid \Phi^{\widetilde{i}}) \\
L_* &= (\nu\,\widetilde{c})(\Omega^{\widetilde{k}} \mid \langle\!\langle Q \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Omega^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \mid \Phi^{\widetilde{i}}) \\
K_1 &= \Theta^{\widetilde{k}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j}} \mid P \\
L_1 &= \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{j}} \mid Q
\end{aligned}$$

Suppose $K_* \xrightarrow{\alpha_l} K'$. The proof is by cases on α_l, where we only consider the subcases with $l \in \Delta$ as the others follow from the definition of domain bisimulation.

For each case we use Lemma 5.3.3, and pattern matching between the syntax of the terms above and of the terms in the lemma. We start with the case for input transitions, which is the easiest.

- $(K_* \xrightarrow{l \cdot k(\widetilde{\pi}\sigma)} K')$: We aim to bring the script instantiated by the transition inside the leftmost merge operator in K_*. In order to do so, we need to avoid both the capture of private channel names in σ by $\widetilde{c}$, and clashes between trigger names in sigma and the vectors $\widetilde{h}, \widetilde{m}$. We split the proof in two cases, depending on whether $k \in \widetilde{k}$ or $k \in \widetilde{i}$.

  - $\Theta^{\widetilde{k}} \xrightarrow{l \cdot k(\widetilde{\pi}\sigma)} \Theta^{\widetilde{k}} \mid \boldsymbol{P}_k\sigma$, where $\langle k \Leftarrow (\widetilde{\pi})\boldsymbol{P}_k \rangle$ is in $\Theta^{\widetilde{k}}$. To avoid clashes between trigger names, we choose for some fresh $\widetilde{h'}, \widetilde{m'}$ and, using standard properties of substitution, rewrite

    $$\langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle = \langle\!\langle P\{\widetilde{m'}/\widetilde{m}\}\{\widetilde{h'}/\widetilde{h}\} \mid \Theta^{\widetilde{m} \frown \widetilde{m'}} \mid \Phi^{\widetilde{h} \frown \widetilde{h'}} \rangle\!\rangle$$

    Let $\rho = \{\widetilde{m'}/\widetilde{m}\}\{\widetilde{h'}/\widetilde{h}\}\{\widetilde{c'}/\widetilde{c}\}$ for a fresh $\widetilde{c'}$, and recall that the private channel names $\widetilde{c}$ cannot appear free in definitions (by well-formedness of scripts). By α-conversion,

    $$K' \equiv (\nu\,\widetilde{c'})(\Theta^{\widetilde{k}} \mid \langle\!\langle \boldsymbol{P}_k\sigma \mid P\rho \mid \Theta^{\widetilde{m} \frown \widetilde{m'}} \mid \Phi^{\widetilde{h} \frown \widetilde{h'}} \rangle\!\rangle \mid \langle\!\langle \Theta^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R\{\widetilde{c'}/\widetilde{c}\} \rangle\!\rangle \mid \Phi^{\widetilde{i}})$$

By $K_1 \approx_\Delta L_1$ and Lemma 5.3.8, $K_2 = K_1\rho \approx_\Delta L_1\rho = L_2$. Since $\Theta^{\widetilde{k}}$ occurs in K_2, $K_2 \xrightarrow{l\cdot k(\widetilde{\pi}\sigma)} K_2 \,|\, \boldsymbol{P}_k\sigma = K_2'$. By $K_2 \approx_\Delta L_2$, $L_2 \xrightarrow{\tau}\!\!\!\!\rightarrow \xrightarrow{l\cdot k(\widetilde{\pi}\sigma)} \xrightarrow{\tau}\!\!\!\!\rightarrow \Omega^{\widetilde{k}} \,|\, \Omega^{\widetilde{m}\curvearrowleft\widetilde{m'}} \,|\, \Omega^{\widetilde{j}} \,|\, Q' = L_2'$ with $L_2' \approx_\Delta K_2'$. We will now use the transition between L_2 and L_2' to derive an appropriate one between L_* and L'. Since the action $\xrightarrow{l\cdot k(\widetilde{\pi}\sigma)}$ necessarily originated by $\Omega^{\widetilde{k}}$, which contains the definition $\langle k \Leftarrow (\widetilde{\pi})\boldsymbol{Q}_k\rangle$, we can reorder the reduction obtaining

$$L_2 \xrightarrow{l\cdot k(\widetilde{\pi}\sigma)} \Omega^{\widetilde{k}} \,|\, \Omega^{\widetilde{m}\curvearrowleft\widetilde{m'}} \,|\, \Omega^{\widetilde{j}} \,|\, Q\rho \,|\, \boldsymbol{Q}_k\sigma \xrightarrow{\tau}\!\!\!\!\rightarrow \Omega^{\widetilde{k}} \,|\, \Omega^{\widetilde{m}\curvearrowleft\widetilde{m'}} \,|\, \Omega^{\widetilde{j}} \,|\, Q'$$

By α-conversion, properties of substitutions and definition of lts,

$$L_* \xrightarrow{l\cdot k(\widetilde{\pi}\sigma)} L''$$

$$L'' = (\nu\, \widetilde{c'})(\Omega^{\widetilde{k}} \,|\, \langle\!\langle \boldsymbol{Q}_k\sigma \,|\, Q\rho \,|\, \Omega^{\widetilde{m}\curvearrowleft\widetilde{m'}} \,|\, \Phi^{\widetilde{h}\curvearrowleft\widetilde{h'}}\rangle\!\rangle \,|\, \langle\!\langle \Omega^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R\{\widetilde{c'}/\widetilde{c}\}\rangle\!\rangle \,|\, \Phi^{\widetilde{i}})$$

where we have brought $\boldsymbol{Q}_k$ inside the leftmost merge operator, in order to preserve the general structure that we have imposed on terms in $\dot{\approx}_\Delta$. By syntactical reasoning, it must be the case that $\boldsymbol{Q}_k\sigma \,|\, Q\rho \xrightarrow{\tau}\!\!\!\!\rightarrow Q'$. By point 2 of Lemma 5.3.12,

$$L'' \xrightarrow{\tau}\!\!\!\!\rightarrow (\nu\, \widetilde{c'})(\Omega^{\widetilde{k}} \,|\, \langle\!\langle Q' \,|\, \Omega^{\widetilde{m}\curvearrowleft\widetilde{m'}} \,|\, \Phi^{\widetilde{h}\curvearrowleft\widetilde{h'}}\rangle\!\rangle \,|\, \langle\!\langle \Omega^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R\{\widetilde{c'}/\widetilde{c}\}\rangle\!\rangle \,|\, \Phi^{\widetilde{i}}) = L'$$

and we conclude because, since $K_2' \approx_\Lambda L_2'$, we have $(K', L') \in \dot{\approx}_\Delta$.

– $\Phi^{\widetilde{i}} \xrightarrow{l\cdot k(\widetilde{\pi}\sigma)} \Phi^{\widetilde{i}} \,|\, \boldsymbol{R}_k\sigma$: similar to the previous case but simpler, since, instead of using the hypothesis $K_1 \approx_\Delta L_1$, it is enough to use syntactical reasoning.

- $(K_* \xrightarrow{l\cdot a(\widetilde{v})} K')$: Similar to the previous case.

- $(K_* \xrightarrow{(\widetilde{b_*},\widetilde{k_*})\overline{l\cdot a}\langle\widetilde{v}\rangle} K')$: We distinguish two cases, depending on whether the output transition is originated by R or P.

  – Suppose $\langle\!\langle \Theta^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R\rangle\!\rangle \xrightarrow{(\widetilde{b'},\widetilde{k'},\widetilde{i'})\overline{l\cdot a}\langle\widetilde{v}\rangle} K_0$, where $\widetilde{c} = \widetilde{c'}, \widetilde{b}$ and $\widetilde{b_*} = \widetilde{b}, \widetilde{b'}$. We assume that the trigger names $\widetilde{k'}$ come from $\Theta^{\widetilde{j}}$, whereas the $\widetilde{i'}$ come from $\Phi^{\widetilde{n}}$ or R. Unfolding the definition of merge, by Lemma 5.3.3 we have that $\langle\!\langle \Theta^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R\rangle\!\rangle = R^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \equiv (\nu\, \widetilde{b'})(\overline{l\cdot a}\langle\widetilde{v_1}^{\,\Theta^{\widetilde{j}}}\rangle \,|\, R_1^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}})$ with $\{\widetilde{b'}\} \subseteq \mathit{fn}(\widetilde{v_1})$, where $\widetilde{v_1} = \widetilde{v_*}^{\,\Phi^{\widetilde{n}}}$ for some appropriate $\widetilde{v_*}$. Moreover, we have $K_0 \equiv R_1^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \,|\, \Theta^{\widetilde{k'}} \,|\, \Phi^{\widetilde{i'}}$, where $\mathfrak{X}(\widetilde{v_1}^{\,\Theta^{\widetilde{j}}}) = (\widetilde{v}; \Theta^{\widetilde{k'}} \,|\, \Phi^{\widetilde{i'}})$. To find out how to split the definitions produced by the extraction into $\Theta^{\widetilde{k'}}$ and $\Phi^{\widetilde{i'}}$, we assume to have first applied the extraction $\mathfrak{X}(\widetilde{v_1}) = (\widetilde{v'}; \Phi^{\widetilde{i'}})$, which ensures that the definitions in $\Phi^{\widetilde{i'}}$ come from R or $\Phi^{\widetilde{n}}$. Then, applying $\mathfrak{X}(\widetilde{v_1}^{\,\Theta^{\widetilde{j}}}) = (\widetilde{v}; \Theta^{\widetilde{k'}} \,|\, \Phi^{\widetilde{i'}})$ we can infer that for each $\langle k' \Leftarrow P_0\rangle$ in $\Theta^{\widetilde{k'}}$ there is $\langle j \Leftarrow P_0\rangle$ in

$\Theta^{\widetilde{j}}$, since, by points 1 and 2 of Lemma 5.3.11, we have that $\widetilde{v}\{\widetilde{k'}/\widetilde{j'}\} = \widetilde{v'}$, where $\widetilde{j'}$ are the triggers in $\widetilde{j}$ occurring also in v_1. With this information, by definition of lts, we can rearrange K' to respect our general pattern

$$K' \equiv (\nu\, \widetilde{c'})(\Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k'}} \mid \langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Theta^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R_1 \rangle\!\rangle \mid \Phi^{\widetilde{i}} \mid \Phi^{\widetilde{i'}})$$

By applying the same argument to L_*, $\langle\!\langle \Omega^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle = R^{\Omega^{\widetilde{j}}\Phi^{\widetilde{n}}}$ and

$$R^{\Omega^{\widetilde{j}}\Phi^{\widetilde{n}}} \equiv (\nu\, \widetilde{b'})(\overline{l{\cdot}a}\langle \widetilde{v_1}^{\Omega^{\widetilde{j}}} \rangle \mid R_1^{\Omega^{\widetilde{j}}\Phi^{\widetilde{n}}}) \xrightarrow{(\widetilde{b'},\widetilde{k'},\widetilde{i'})\overline{l{\cdot}a}\langle\widetilde{v}\rangle} R_1^{\Omega^{\widetilde{j}}\Phi^{\widetilde{n}}} \mid \Omega^{\widetilde{k'}} \mid \Phi^{\widetilde{i'}}$$

for $\mathfrak{X}(\widetilde{v_1}^{\Omega^{\widetilde{j}}}) = (\widetilde{v}; \Omega^{\widetilde{k'}} \mid \Phi^{\widetilde{i'}})$, where for each $\langle k' \Leftarrow Q_0 \rangle$ in $\Omega^{\widetilde{k'}}$ there is a $\langle j \Leftarrow Q_0 \rangle$ in $\Omega^{\widetilde{j}}$. By definition of lts,

$$L_* \xrightarrow{(\widetilde{b},\widetilde{b'},\widetilde{k'},\widetilde{i'})\overline{l{\cdot}a}\langle\widetilde{v}\rangle} L' \equiv (\nu\, \widetilde{c'})(\Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} \mid \langle\!\langle Q \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Omega^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R_1 \rangle\!\rangle \mid \Phi^{\widetilde{i}} \mid \Phi^{\widetilde{i'}})$$

By $K_1 \approx_\Delta L_1$ and point 2 of Lemma 5.3.13,

$$\Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j}} \mid P \approx_\Delta \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{j}} \mid Q$$

and we conclude because $(K', L') \in \dot{\approx}_\Delta$.

– Suppose $\langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \xrightarrow{(\widetilde{b'},\widetilde{k'},\widetilde{m'},\widetilde{i'})\overline{l{\cdot}a}\langle\widetilde{v}\rangle} K_0$, where $\widetilde{c} = \widetilde{c'}, \widetilde{b}$ and $\widetilde{b}_* = \widetilde{b}, \widetilde{b'}$. We assume $\widetilde{i'}$ are the new trigger names from $\Phi^{\widetilde{h}}$, $\widetilde{m'}$ the ones from $\Theta^{\widetilde{m}}$ and $\widetilde{k'}$ the ones from P. Differently from the previous case, we need to keep track explicitly of the triggers coming from $\Theta^{\widetilde{m}}$, which will correspond in L_* to triggers coming from $\Omega^{\widetilde{m}}$. We have that

$$\langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle = P^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \equiv (\nu\, \widetilde{b'})(\overline{l{\cdot}a}\langle \widetilde{v_1}^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \rangle \mid P_1^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}})$$

with $\{\widetilde{b'}\} \subseteq \mathit{fn}(\widetilde{v})$, and $K_0 \equiv P_1^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \mid \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m'}} \mid \Phi^{\widetilde{i'}}$, where we assume that $\widetilde{k'}$, $\widetilde{m'}$ and $\widetilde{i'}$ are disjoint from $\widetilde{k}$, $\widetilde{m}$ and $\widetilde{i}$. Moreover, we have

$$\mathfrak{X}(\widetilde{v_1}^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}}) = (\widetilde{v}; \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m'}} \mid \Phi^{\widetilde{i'}})$$
$$\mathfrak{X}(\widetilde{v_1}^{\Theta^{\widetilde{m}}}) = (\widetilde{v''}; \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m'}})$$
$$\mathfrak{X}(\widetilde{v_1}) = (\widetilde{v'}; \Theta^{\widetilde{k'}})$$

Hence, by Lemma 5.3.11, for each $\langle i' \Leftarrow R_0 \rangle$ in $\Phi^{\widetilde{i'}}$ there is $\langle h \Leftarrow R_0 \rangle$ in $\Phi^{\widetilde{h}}$, and for each $\langle m' \Leftarrow P_0 \rangle$ in $\Theta^{\widetilde{m'}}$ there is $\langle m \Leftarrow P_0 \rangle$ in $\Theta^{\widetilde{m}}$. By definition of lts,

$$K' \equiv (\nu\, \widetilde{c'})(\Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m'}} \mid \langle\!\langle P_1 \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Theta^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \mid \Phi^{\widetilde{i}} \mid \Phi^{\widetilde{i'}})$$

where we have used the information gathered above on $\Theta^{\widetilde{k'}}$, $\Theta^{\widetilde{m'}}$ and $\Phi^{\widetilde{i'}}$ to decide how to rearrange K', in order to fit our general pattern. By definition of lts,

$$K_1 \xrightarrow{(\widetilde{b'},\widetilde{k'})\overline{l{\cdot}a}\langle\widetilde{v'}\rangle} P_1 \mid \Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j}} = K_1'$$

where $\{\widetilde{b'}\} \subseteq fn(\widetilde{v'})$ and $\mathfrak{X}(\widetilde{v_1}) = (\widetilde{v'}; \Theta^{\widetilde{k'}})$. By $K_1 \approx_\Delta L_1$ and point 3 of Lemma 5.3.12,

$$L_1 \xrightarrow{(\widetilde{b'},\widetilde{k'})\overline{l\cdot a}\langle\widetilde{v'}\rangle} \twoheadrightarrow Q_1 \mid \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{j}} = L_1'$$

and $K_1' \approx_\Delta L_1'$. We now derive a corresponding transition for L_*. Since none of the transitions above can be generated by a definition, we can deduce

$$Q \xrightarrow{\tau} \twoheadrightarrow (\nu\, \widetilde{b'})(\overline{l\cdot a}\langle\widetilde{v_2}\rangle \mid Q_2) = Q_3, \quad Q_3 \xrightarrow{(\widetilde{b'},\widetilde{k'})\overline{l\cdot a}\langle\widetilde{v'}\rangle} Q_2 \mid \Omega^{\widetilde{k'}} \xrightarrow{\tau} \twoheadrightarrow Q_1 \mid \Omega^{\widetilde{k'}}$$

where $\mathfrak{X}(\widetilde{v_2}) = (\widetilde{v'}; \Omega^{\widetilde{k'}})$. By Lemma 5.3.11, $\widetilde{v_2}$ has exactly the same occurrences of trigger names in $\widetilde{h}$ as $\widetilde{v'}$, which are the same of $\widetilde{v_1}$. By point 2 of Lemma 5.3.12, $\langle\!\langle Q \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \xrightarrow{\tau} \twoheadrightarrow \langle\!\langle Q_3 \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle$. By syntactical reasoning

$$\langle\!\langle Q_3 \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \xrightarrow{(\widetilde{b'},\widetilde{k'},\widetilde{m'},\widetilde{i'})\overline{l\cdot a}\langle\widetilde{v}\rangle} \langle\!\langle Q_2 \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m'}} \mid \Phi^{\widetilde{i'}}$$

where $\mathfrak{X}(\widetilde{v_2}^{\Omega^{\widetilde{m}}\Phi^{\widetilde{h}}}) = (\widetilde{v}; \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m'}} \mid \Phi^{\widetilde{i'}})$. By point 2 of Lemma 5.3.12 and by definition of lts,

$$\langle\!\langle Q_2 \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m'}} \mid \Phi^{\widetilde{i'}} \xrightarrow{\tau} \twoheadrightarrow \langle\!\langle Q_1 \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m'}} \mid \Phi^{\widetilde{i'}}$$

By definition of lts, $L_* \xrightarrow{(\widetilde{b},\widetilde{b'},\widetilde{k'},\widetilde{m'},\widetilde{i'})\overline{l\cdot a}\langle\widetilde{v}\rangle} \twoheadrightarrow L'$ and

$$L' \equiv (\nu\, \widetilde{c'})(\Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m'}} \mid \langle\!\langle Q_1 \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Omega^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \mid \Phi^{\widetilde{i}} \mid \Phi^{\widetilde{i'}})$$

By $K_1' \approx_\Delta L_1'$ and point 2 of Lemma 5.3.13,

$$\Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m'}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j}} \mid P_1 \approx_\Delta \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m'}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{j}} \mid Q_1$$

and we conclude because $(K', L') \in \dot{\approx}_\Delta$.

- $(K_* \xrightarrow{(\widetilde{b_*},\widetilde{k_*})\overline{l\cdot j}\langle\widetilde{v}\rangle} K')$: Analogous to the case for output.
- $(K_* \xrightarrow{(\widetilde{k})l\cdot \mathsf{req}\langle p\rangle(T)} K')$: By combining the argument for input and output.
- $(K_* \xrightarrow{l\cdot\tau} K')$: First we analyze the case where the transition is determined by the interaction of R and P, then the case where the transition is derived by R or P in isolation.

  Interaction. We analyze the transitions resulting from an interaction between R and P. We must distinguish four cases depending on whether R or P receives the value, and whether the value is received by a replicated input. Suppose $\langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid \langle\!\langle \Theta^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R \rangle\!\rangle \xrightarrow{\tau} K_0$.

Replicated input by R: By Lemma 5.3.3, we have that

$$R^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \equiv (\nu\, \widetilde{b})(R_1^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \mid !l{\cdot}a(\widetilde{\pi}).R_2^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}})$$

where $\widetilde{b}$ is fresh with respect to P and $\widetilde{c}$, and $a \notin \{\widetilde{b}\}$. We also have that $P^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \equiv (\nu\, \widetilde{c'})(\overline{l{\cdot}a}\langle \widetilde{v}^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}}\rangle \mid P_1^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}})$, where $\widetilde{c'}$ is fresh with respect to R and $\widetilde{c}$, $\{\widetilde{c'}\} \subseteq \mathit{fn}(v^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}})$, and $\widetilde{v}^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} = \widetilde{\pi}\sigma'$. Moreover,

$$K_0 \equiv (\nu\, \widetilde{c'})(P_1^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \mid (\nu\, \widetilde{b})(R_1^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \mid !l{\cdot}a(\widetilde{\pi}).R_2^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \mid R_2^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}}\sigma'))$$

Since scripts cannot contain free private names, $\{\widetilde{c'}\} \subseteq \mathit{fn}(v^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}})$ implies $\{\widetilde{c'}\} \subseteq \mathit{fn}(v)$. Since patterns cannot contain scripts (or trigger names), there exists σ such that $\widetilde{v} = \widetilde{\pi}\sigma$ and $\sigma^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} = \sigma'$. We want to derive a configuration K' of the right form. By definition of merge, $P_1^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} = \langle\!\langle P_1 \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}}\rangle\!\rangle$. Before rewriting

$$R_* = (\nu\, \widetilde{b})(R_1^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \mid !l{\cdot}a(\widetilde{\pi}).R_2^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}} \mid R_2^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}}\sigma^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}})$$

in terms of the merge operator, we want to to be explicit about the scripts occurring in $\widetilde{v}$. Suppose $\mathfrak{X}(\widetilde{v}) = (\widetilde{v'}; \Theta^{\widetilde{k'}})$ for fresh $\widetilde{k'}$. Since $\widetilde{v} = \widetilde{\pi}\sigma$ and $\widetilde{v'}$ differs from $\widetilde{v}$ only for having triggers replacing scripts, there exists ρ such that $\widetilde{v'} = \widetilde{\pi}\rho$. By Lemma 5.3.11, $\widetilde{v} = \widetilde{v'}^{\Theta^{\widetilde{k'}}}$. Since $\widetilde{v} = \widetilde{\pi}\sigma$, $\widetilde{v'} = \widetilde{\pi}\rho$ and $\widetilde{\pi}$ cannot contain trigger names, we have that $\sigma = \rho^{\Theta^{\widetilde{k'}}}$, and both $\widetilde{v}$ and $\widetilde{v'}$ have the same occurrences of triggers in $\widetilde{h}$ and $\widetilde{m}$. Without loss of generality, we assume that $\{\widetilde{j}, \widetilde{n}\} \cap \mathit{fn}(\rho) = \emptyset$. By standard properties of substitution, $R_2^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}}\sigma^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} = (R_2\rho\{\widetilde{m'}/\widetilde{m}\}^{\Theta^{\widetilde{m}\frown\widetilde{m'}}\Phi^{\widetilde{h}}\Theta^{\widetilde{k'}}})^{\Theta^{\widetilde{j}}\Phi^{\widetilde{n}}}$, where the vector $\widetilde{m'}$ is fresh. By definition of merge,

$$R_* = \langle\!\langle \Theta^{\widetilde{j}} \mid \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m}\frown\widetilde{m'}} \mid \Phi^{\widetilde{n}} \mid (\nu\, \widetilde{b})(R_1 \mid !l{\cdot}a(\widetilde{\pi}).R_2 \mid R_2\rho\{\widetilde{m'}/\widetilde{m}\}^{\Phi^{\widetilde{h}}})\rangle\!\rangle$$

By definition of lts,

$$K' \equiv (\nu\, \widetilde{c}, \widetilde{c'})(\Theta^{\widetilde{k}} \mid \langle\!\langle P_1 \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}}\rangle\!\rangle \mid R_* \mid \Phi^{\widetilde{i}})$$

By definition of lts,

$$K_1 \xrightarrow{(\widetilde{c'},\widetilde{k'})\overline{l{\cdot}a}\langle\widetilde{v'}\rangle} P_1 \mid \Theta^{\widetilde{j}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k'}} = K_1'$$

where, as noted above, $\mathfrak{X}(\widetilde{v}) = (\widetilde{v'}; \Theta^{\widetilde{k'}})$. By $K_1 \approx_\Lambda L_1$,

$$L_1 \xrightarrow{\tau} \!\!\!\!\rightarrow (\nu\, \widetilde{c'})(Q_1 \mid \overline{l{\cdot}a}\langle\widetilde{v_2}\rangle) \mid \Omega^{\widetilde{j}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{k}} = L_2$$

$$L_2 \xrightarrow{(\widetilde{c'},\widetilde{k'})\overline{l{\cdot}a}\langle\widetilde{v'}\rangle} Q_1 \mid \Omega^{\widetilde{j}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} = L_3$$

where $\mathfrak{X}(\widetilde{v_2}) = (\widetilde{v'}; \Omega^{\widetilde{k'}})$, and

$$L_3 \xrightarrow{\tau} Q' \mid \Omega^{\widetilde{j}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} = L_1'$$

with $K_1' \approx_\Lambda L_1'$. By point 2 of Lemma 5.3.12, since $Q \xrightarrow{\tau} (\nu\, \widetilde{c'})(Q_1 \mid \overline{l{\cdot}a}\langle\widetilde{v_2}\rangle)$,

$$\langle\!\langle Q \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \xrightarrow{\tau} \langle\!\langle (\nu\, \widetilde{c'})(Q_1 \mid \overline{l{\cdot}a}\langle\widetilde{v_2}\rangle) \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle = (\nu\, \widetilde{c'})(Q_1^{\Phi^{\widetilde{h}}} \mid \overline{l{\cdot}a}\langle \widetilde{v_2}^{\Omega^{\widetilde{m}}\Phi^{\widetilde{h}}} \rangle)$$

By syntactical reasoning, $L_* \xrightarrow{\tau} \xrightarrow{\tau} L''$, where

$$L'' = (\nu\, \widetilde{c}, \widetilde{c'})(\Omega^{\widetilde{k}} \mid \langle\!\langle Q_1 \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid R_*' \mid \Phi^{\widetilde{i}})$$

where, using an argument similar to that used for R_*,

$$R_*' = \langle\!\langle \Omega^{\widetilde{j}} \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m} \frown \widetilde{m'}} \mid \Phi^{\widetilde{n}} \mid (\nu\, \widetilde{b})(R_1 \mid !l{\cdot}a(\widetilde{\pi}).R_2 \mid R_2\rho\{\widetilde{m'}/\widetilde{m}\}^{\Phi^{\widetilde{h}}}) \rangle\!\rangle$$

Note that R_*' is essentially R_* where each Θ is replaced by an Ω. By point 2 of Lemma 5.3.12, since $Q_1 \xrightarrow{\tau} Q'$ we have that $L'' \xrightarrow{\tau} L'$, where

$$L' \equiv (\nu\, \widetilde{c}, \widetilde{c'})(\Omega^{\widetilde{k}} \mid \langle\!\langle Q' \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \rangle\!\rangle \mid R_*' \mid \Phi^{\widetilde{i}})$$

By $K_1' \approx_\Delta L_1'$ and point 2 of Lemma 5.3.13,

$$\Theta^{\widetilde{k}} \mid \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{m} \frown \widetilde{m'}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j}} \mid P_1 \approx_\Delta \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{m} \frown \widetilde{m'}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{j}} \mid Q_1$$

and we conclude because $(K', L') \in \dot{\approx}_\Delta$.

Input by R: analogous to the previous case.

Input by P: By Lemma 5.3.3, we have $R^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \equiv (\nu\, \widetilde{c'})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \mid \overline{l{\cdot}a}\langle\widetilde{v}^{\Theta^{\widetilde{j}}}\rangle)$, where $\widetilde{c'}$ is fresh and $\{\widetilde{c'}\} \subseteq \mathit{fn}(\widetilde{v})$. Moreover, $P^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \equiv (\nu\, \widetilde{b})(P_1^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \mid l{\cdot}a(\widetilde{\pi}).P_2^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}})$, where $\widetilde{b}$ is fresh and $\widetilde{v} = \widetilde{\pi}\sigma$ (since scripts and trigger names cannot appear in patterns). Additionally, $K_0 \equiv (\nu\, \widetilde{c'})((\nu\, \widetilde{b})(P_1^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}} \mid P_2^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}}\sigma^{\Theta^{\widetilde{j}}}) \mid R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}})$. In order to derive a K' of a suitable form, we follow a strategy similar to the one used in the case of replicated input by R. Let $\widetilde{v'} = \widetilde{v}\{\widetilde{j'}/\widetilde{j}\}$ for some vector of fresh triggers $\widetilde{j'}$. Since $\widetilde{v} = \widetilde{\pi}\sigma$ and $\widetilde{\pi}$ cannot contain triggers, $\widetilde{v'} = \widetilde{\pi}\sigma\{\widetilde{j'}/\widetilde{j}\}$. By standard properties of substitution, $\widetilde{v}^{\Theta^{\widetilde{j}}} = \widetilde{v'}^{\Theta^{\widetilde{j} \frown \widetilde{j'}}}$. Suppose $\mathfrak{X}(\widetilde{v'}) = (\widetilde{v_1}; \Phi^{\widetilde{h'}})$ for some fresh $\widetilde{h'}$ such that $\widetilde{v_1} = \widetilde{\pi}\rho$. By Lemma 5.3.11 and by freshness of $\widetilde{j'}, \widetilde{h'}$, we have $\widetilde{v'} = \widetilde{v_1}^{\Phi^{\widetilde{h'}}}$, $\rho = \sigma\{\widetilde{j'}/\widetilde{j}\}^{\Phi^{\widetilde{h'}}}$ and $P_2^{\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}}\sigma^{\Theta^{\widetilde{j}}} = P_2\rho^{\Theta^{\widetilde{j} \frown \widetilde{j'}}\Phi^{\widetilde{h'}}\Theta^{\widetilde{m}}\Phi^{\widetilde{h}}}$. By definition of lts,

$$K' \equiv (\nu\, \widetilde{c}, \widetilde{c'})(\Theta^{\widetilde{k}} \mid \langle\!\langle (\nu\, \widetilde{b})(P_1 \mid P_2\rho) \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j} \frown \widetilde{j'}} \mid \Phi^{\widetilde{h}} \mid \Phi^{\widetilde{h'}} \rangle\!\rangle \mid \langle\!\langle \Theta^{\widetilde{j}} \mid \Phi^{\widetilde{n}} \mid R_1 \rangle\!\rangle \mid \Phi^{\widetilde{i}})$$

By definition of lts,

$$K_1 \xrightarrow{l{\cdot}a(\widetilde{v_1})} \xrightarrow{\tau} (\nu\, \widetilde{b})(P_1 \mid P_2\rho) \mid \Omega^{\widetilde{j}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{k}} = K_1'$$

By $K_1 \approx_\Lambda L_1$ and composing the weak actions of L_1,

$$L_1 \xrightarrow{l\cdot a(\widetilde{v_1})} \!\!\!\rightarrow Q' \,|\, \Omega^{\widetilde{j}} \,|\, \Omega^{\widetilde{m}} \,|\, \Omega^{\widetilde{k}} = L_1'$$

and $K_1' \approx_\Lambda L_1'$. By syntactical reasoning,

$$Q \xrightarrow{\tau} \!\!\!\rightarrow Q_1 \xrightarrow{l\cdot a(\widetilde{v_1})} Q_1 \,|\, \overline{l\cdot a}\langle\widetilde{v_1}\rangle \xrightarrow{\tau} \!\!\!\rightarrow Q'$$

By syntactical reasoning and by point 2 of Lemma 5.3.12,

$$L_* \xrightarrow{\tau} \!\!\!\rightarrow (\nu\,\widetilde{c})(\Omega^{\widetilde{k}} \,|\, \langle\!\langle Q_1 \,|\, \Omega^{\widetilde{m}} \,|\, \Phi^{\widetilde{h}}\rangle\!\rangle \,|\, \langle\!\langle \Omega^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R\rangle\!\rangle \,|\, \Phi^{\widetilde{i}}) = L''$$

By structural congruence, by $\widetilde{v'} = \widetilde{v_1}^{\Phi^{\widetilde{h'}}}$ and $\widetilde{v}^{\Omega^{\widetilde{j}}} = \widetilde{v'}^{\Omega^{\widetilde{j}} \frown \widetilde{j'}}$, and since $R^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}} \equiv (\nu\,\widetilde{c'})(R_1^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}} \,|\, \overline{l\cdot a}\langle\widetilde{v}^{\Omega^{\widetilde{j}}}\rangle)$,

$$L'' \equiv (\nu\,\widetilde{c},\widetilde{c'})(\Omega^{\widetilde{k}} \,|\, \langle\!\langle Q_1 \,|\, \overline{l\cdot a}\langle\widetilde{v_1}\rangle \,|\, \Omega^{\widetilde{m}} \,|\, \Omega^{\widetilde{j} \frown \widetilde{j'}} \,|\, \Phi^{\widetilde{h}} \,|\, \Phi^{\widetilde{h'}}\rangle\!\rangle \,|\, \langle\!\langle \Omega^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R_1\rangle\!\rangle \,|\, \Phi^{\widetilde{i}})$$

By point 2 of Lemma 5.3.12, $L'' \xrightarrow{\tau} \!\!\!\rightarrow L'$ where

$$L' = (\nu\,\widetilde{c},\widetilde{c'})(\Omega^{\widetilde{k}} \,|\, \langle\!\langle Q' \,|\, \Omega^{\widetilde{m}} \,|\, \Omega^{\widetilde{j} \frown \widetilde{j'}} \,|\, \Phi^{\widetilde{h}} \,|\, \Phi^{\widetilde{h'}}\rangle\!\rangle \,|\, \langle\!\langle \Omega^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R_1\rangle\!\rangle \,|\, \Phi^{\widetilde{i}})$$

By point 2 of Lemma 5.3.13, $K_1' \,|\, \Theta^{\widetilde{j} \frown \widetilde{j'}} \approx_\Lambda L_1' \,|\, \Omega^{\widetilde{j} \frown \widetilde{j'}}$. and we conclude because $(K', L') \in \dot{\approx}_\Delta$.

Replicated input by P: similar to the previous case.

Isolation. We show the case for R, as the case for P is similar. There are four ways to derive the sub-transition $\langle\!\langle \Theta^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R\rangle\!\rangle \xrightarrow{l\cdot\tau} M_1$.

- $R^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \equiv (\nu\,\widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \,|\, l\cdot c(\widetilde{\pi}).R_2^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \,|\, \overline{l\cdot c}\langle\widetilde{\pi}\sigma^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}\rangle)$ and

  $$M_1 \equiv (\nu\,\widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \,|\, (R_2^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}})\sigma^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}) \equiv \langle\!\langle \Theta^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, (\nu\,\widetilde{a})(R_1 \,|\, R_2\sigma)\rangle\!\rangle$$

  By syntactical reasoning, we can also derive

  $$\langle\!\langle \Omega^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, R\rangle\!\rangle \equiv (\nu\,\widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}} \,|\, l\cdot c(\widetilde{\pi}).R_2^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}} \,|\, \overline{l\cdot c}\langle\widetilde{\pi}\sigma^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}}\rangle)$$

  $$\xrightarrow{l\cdot\tau} (\nu\,\widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}} \,|\, (R_2^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}})\sigma^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}}) = \langle\!\langle \Omega^{\widetilde{j}} \,|\, \Phi^{\widetilde{n}} \,|\, (\nu\,\widetilde{a})(R_1 \,|\, R_2\sigma)\rangle\!\rangle$$

  and we conclude because, by definition of lts, we can use this transition to derive a transition for L_* matching the one of K_*, with the resulting states K' and L' still in $\dot{\approx}_\Delta$.
- $R^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \equiv (\nu\,\widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \,|\, !l\cdot c(\widetilde{\pi}).R_2^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \,|\, \overline{l\cdot c}\langle\widetilde{\pi}\sigma^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}\rangle)$: analogous to the previous case.
- $R^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \equiv (\nu\,\widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \,|\, m\cdot\mathsf{go}\ l.R_2^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}})$: similar to the previous cases, although the reasoning on $R_2^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}$ is carried on at location l.

– $R^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \equiv (\nu\, \widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \mid (x,\widetilde{\pi})R_2 \circ \langle l, \widetilde{\pi}\sigma^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}\rangle)$ and

$$M_1 \equiv (\nu\, \widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \mid R_2\{{}^{l}/_{x}\}\sigma^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}})$$

where we have used the equation $R_2^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} = R_2$, which holds because scripts cannot contain trigger names. In order for $R^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}$ to have the form given above, R itself must have one of the three forms given below.

1. If $R \equiv (\nu\, \widetilde{a})(R_1 \mid (x,\widetilde{\pi})R_2 \circ \langle l, \widetilde{\pi}\sigma\rangle)$ then, by definition of script, $((x,\widetilde{\pi})R_2 \circ \langle l, \widetilde{\pi}\sigma\rangle)^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} = (x,\widetilde{\pi})R_2 \circ \langle l, \widetilde{\pi}\sigma^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}\rangle$ and the reasoning is similar to the cases above.
2. If $R \equiv (\nu\, \widetilde{a})(R_1 \mid k \circ \langle l, \widetilde{\pi}\sigma\rangle)$ where $\Phi^{\widetilde{n}} = \Phi^{\widetilde{n_1}} \mid \langle k \Leftarrow (x,\widetilde{\pi})R_2\rangle \mid \Phi^{\widetilde{n_2}}$ with $\widetilde{n_1}, k, \widetilde{n_2} = \widetilde{n}$, then we have

$$R^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \equiv (\nu\, \widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}} \mid (x,\widetilde{\pi})R_2 \circ \langle l, \widetilde{\pi}\sigma^{\Phi^{\widetilde{n}}\Theta^{\widetilde{j}}}\rangle)$$
$$R^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}} \equiv (\nu\, \widetilde{a})(R_1^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}} \mid (x,\widetilde{\pi})R_2 \circ \langle l, \widetilde{\pi}\sigma^{\Phi^{\widetilde{n}}\Omega^{\widetilde{j}}}\rangle)$$

and the reasoning is once again analogous to case 1.
3. The last and most interesting case arises if $R \equiv (\nu\, \widetilde{a})(R_1 \mid k \circ \langle l, \widetilde{\pi}\sigma\rangle)$ and $\Theta^{\widetilde{j}} = \Theta^{\widetilde{j_1}} \mid \langle k \Leftarrow (x,\widetilde{\pi})R_2\rangle \mid \Theta^{\widetilde{j_2}}$, where $\widetilde{j_1}, k, \widetilde{j_2} = \widetilde{j}$. In this case, a tau transition by K_* corresponds to a (Lts Def) transition by K_1. Without loss of generality, we assume that the names $\widetilde{a}$ are fresh. In order to derive an appropriate K', we need to be explicit about the scripts in $\widetilde{\pi}\sigma$. Hence, suppose that $\mathfrak{X}(\widetilde{\pi}\sigma) = (\widetilde{\pi}\rho; \Phi^{\widetilde{n'}})$ where $\widetilde{n'}$ is fresh.
   By definition of lts, $K_* \xrightarrow{\tau} K'$ where

$$K' = (\nu\, \widetilde{c})(\Theta^{\widetilde{k}} \mid \langle\!\langle P \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}}\rangle\!\rangle \mid \langle\!\langle (\nu\, \widetilde{a})(R_1 \mid R_2\{{}^{l}/_{x}\}\rho) \mid \Phi^{\widetilde{n'}} \mid \Phi^{\widetilde{n}} \mid \Theta^{\widetilde{j}}\rangle\!\rangle \mid \Phi^{\widetilde{i}})$$

   Since R_2 comes from the definition $\langle k \Leftarrow (x,\widetilde{\pi})R_2\rangle$ which is part of $\Theta^{\widetilde{j}}$ (hence was originated by some previous transition of P), we need to move $R_2R_2\{{}^{l}/_{x}\}\rho$ inside the leftmost merge operator.

$$K' \equiv (\nu\, \widetilde{c}, \widetilde{a})(\Theta^{\widetilde{k}} \mid \langle\!\langle P \mid R_2\{{}^{l}/_{x}\}\rho \mid \Theta^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \mid \Phi^{\widetilde{n'}}\rangle\!\rangle \mid \langle\!\langle R_1 \mid \Phi^{\widetilde{n}} \mid \Theta^{\widetilde{j}}\rangle\!\rangle \mid \Phi^{\widetilde{i}})$$

   With K' of this form, we will derive a transition from K_1 to a suitable K_1', and use the bisimilarity hypothesis to derive a matching transition for L_*. By definition of lts,

$$K_1 \xrightarrow{l \cdot k(\widetilde{\pi}\rho)} \Theta^{\widetilde{k}} \mid \Theta^{\widetilde{m}} \mid \Theta^{\widetilde{j}} \mid P \mid R_2\{{}^{l}/_{x}\}\rho = K_1'$$

   By $K_1 \approx_\Lambda L_1$ and by point 3 of Lemma 5.3.12 we know that $\Omega^{\widetilde{j}} = \Omega^{\widetilde{j_1}} \mid \langle k \Leftarrow (x,\widetilde{\pi}')R_3\rangle \mid \Omega^{\widetilde{j_2}}$. Using our standard reasoning on the hypothesis $K_1 \approx_\Lambda L_1$, we can derive the transition

$$L_1 \xrightarrow{l \cdot k(\widetilde{\pi}\rho)} \Omega^{\widetilde{k}} \mid \Omega^{\widetilde{m}} \mid \Omega^{\widetilde{j}} \mid Q' = L_1' \approx_\Lambda K_1'$$

Breaking down the transition into $\xrightarrow{\tau}\!\!\!\!\rightarrow \xrightarrow{l\cdot k(\widetilde{\pi}\rho)}\!\!\!\!\rightarrow \xrightarrow{\tau}\!\!\!\!\rightarrow$ and using point 2 of Lemma 5.3.12, we obtain

$$L_* \xrightarrow{\tau}\!\!\!\!\rightarrow L' = (\nu\,\widetilde{c},\widetilde{a})(\Omega^{\widetilde{k}} \mid \langle\!\langle Q' \mid \Omega^{\widetilde{m}} \mid \Phi^{\widetilde{h}} \mid \Phi^{\widetilde{n'}} \rangle\!\rangle \mid \langle\!\langle R_1 \mid \Phi^{\widetilde{n}} \mid \Omega^{\widetilde{j}} \rangle\!\rangle \mid \Phi^{\widetilde{i}})$$

and hence $(K', L') \in \dot{\approx}_\Delta$.

□

We can finally prove that domain bisimilarity is a congruence on both configurations and, more importantly, open processes. The extension to open processes does not involve significant difficulties because, by definition, bisimilarity for open processes is closed under arbitrary substitutions.

Theorem 5.3.15 (Congruence) *Domain bisimilarity is a congruence:*

1. *for all configuration contexts $C \in \mathcal{K}_\mathcal{W}$ (Figure 5.1), if $K \approx_\Lambda L$ then $C[K] \approx_\Lambda C[L]$;*
2. *for all full process contexts $C \in \mathcal{K}_\mathrm{f}$ (Definition 4.4.1), if $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q}$ then $C[\boldsymbol{P}] \approx_\Lambda C[\boldsymbol{Q}]$.*

Proof.

1. By Lemma 5.3.10, $K \approx_\Lambda L \implies (\nu\,\widetilde{c})K \approx_\Lambda (\nu\,\widetilde{c})L$. By Lemma 5.3.14, $K \approx_\Lambda L \implies K \mid M \approx_\Lambda L \mid M$. By Proposition 5.3.1, $K \mid M \approx_\Lambda L \mid M \implies M \mid K \approx_\Lambda M \mid L$.

2. We need to show that

   (a) $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q} \implies (\nu\,\widetilde{c})\boldsymbol{P} \approx_\Lambda (\nu\,\widetilde{c})\boldsymbol{Q}$;
   (b) $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q} \implies \boldsymbol{P} \mid \boldsymbol{R} \approx_\Lambda \boldsymbol{Q} \mid \boldsymbol{R}$;
   (c) $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q} \implies \boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P} \approx_\Lambda \boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{Q}$;
   (d) $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q} \implies !\boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P} \approx_\Lambda !\boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{Q}$;
   (e) $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q} \implies \boldsymbol{l}\cdot\mathsf{go}\,\boldsymbol{m}.\boldsymbol{P} \approx_\Lambda \boldsymbol{l}\cdot\mathsf{go}\,\boldsymbol{m}.\boldsymbol{Q}$.

   By definition $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q}$ if and only if $\boldsymbol{P}\sigma \approx_\Lambda \boldsymbol{Q}\sigma$ for all closing substitutions σ.

   (a) Consider an arbitrary closing substitution σ for $\boldsymbol{P}, \boldsymbol{Q}$. Since we assume substitutions to be capture avoiding, $((\nu\,\widetilde{c})\boldsymbol{P})\sigma \equiv (\nu\,\widetilde{c'})(\boldsymbol{P}\{^{c'}/_{c}\}\sigma)$ and $((\nu\,\widetilde{c})\boldsymbol{Q})\sigma \equiv (\nu\,\widetilde{c'})(\boldsymbol{Q}\{^{c'}/_{c}\}\sigma)$. By hypothesis, $\boldsymbol{P}\{^{c'}/_{c}\}\sigma \approx_\Lambda \boldsymbol{Q}\{^{c'}/_{c}\}\sigma$. By point 1 above, $(\nu\,\widetilde{c'})(\boldsymbol{P}\{^{c'}/_{c}\}\sigma) \approx_\Lambda (\nu\,\widetilde{c'})(\boldsymbol{Q}\{^{c'}/_{c}\}\sigma)$.

   (b) Similar to point 2a, using point 1.

   (c) Let the family $\dot{\approx}$ have the generic element

   $$\dot{\approx}_\Delta = \approx_\Delta \cup \{(\boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}\sigma \mid M, \boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{Q}\sigma \mid M) \;\; : \;\; \boldsymbol{P}\sigma \approx_\Delta \boldsymbol{Q}\sigma\}$$

   where σ is a closing substitution, $M = \prod_{n\geq 0} \overline{l_i\cdot c_i}\langle\widetilde{v}_i\rangle$ and $\mathit{scripts}(\widetilde{v}_i) = \emptyset$. The thesis follows by showing that $\dot{\approx}$ is a domain bisimulation, using point 1.

(d) Similar to point 2c, using also transitivity of $\approx_\Delta$ (Lemma 5.3.9).

(e) Let the family $\dot{\approx}$ have the generic element $\dot{\approx}_\Delta$ given by

$$\text{if } m \notin \Delta\text{: } \dot{\approx}_\Delta = \approx_\Delta \cup \{(\boldsymbol{l}{\cdot}\mathsf{go}\,\boldsymbol{m}.\boldsymbol{P}\sigma \mid M, \boldsymbol{l}{\cdot}\mathsf{go}\,\boldsymbol{m}.\boldsymbol{Q}\sigma \mid M) \;\; : \;\; \boldsymbol{P}\sigma \approx_\Delta \boldsymbol{Q}\sigma\}$$

$$\text{if } m \in \Delta\text{: } \dot{\approx}_\Delta = \approx_\Delta \cup \left\{ \begin{array}{r} (\boldsymbol{l}{\cdot}\mathsf{go}\,\boldsymbol{m}.\boldsymbol{P}\sigma \mid M, \boldsymbol{l}{\cdot}\mathsf{go}\,\boldsymbol{m}.\boldsymbol{Q}\sigma \mid M), \\ (\boldsymbol{P}\sigma \mid M, \boldsymbol{l}{\cdot}\mathsf{go}\,\boldsymbol{m}.\boldsymbol{Q}\sigma \mid M), \\ (\boldsymbol{l}{\cdot}\mathsf{go}\,\boldsymbol{m}.\boldsymbol{P}\sigma \mid M, \boldsymbol{Q}\sigma \mid M) \end{array} \;\; : \;\; \boldsymbol{P}\sigma \approx_\Delta \boldsymbol{Q}\sigma \right\}$$

where $\mathit{dom}(M) \subseteq \Delta$, σ is a closing substitution, $M = \prod_{n\geq 0} \overline{l_i{\cdot}c_i}\langle\widetilde{v}_i\rangle$ and $\mathit{scripts}(\widetilde{v}_i) = \emptyset$. The family $\dot{\approx}$ is a domain bisimulation.

□

5.3.3 Soundness

In this section, we show *soundness*: if two processes are domain bisimilar with respect to Λ, then they are request congruent with respect to Λ. Our strategy for proving the soundness of $\approx_\Lambda$ consists of three main steps. First, we define an auxiliary relation $\asymp$ on Core X$d\pi$ networks such that two networks are in the relation if the corresponding processes, in parallel with the definitions extracted from the scripts in the corresponding stores, are Λ-bisimilar. Second we show that $\asymp$ is included in $\sim_r^\Lambda$, and third we use $\asymp$ as a stepping stone to relate $\approx_\Lambda$ with $\sim_r^\Lambda$.

We begin comparing reductions and transitions. If a configuration K can perform a tau transition to become K', then the process $\langle\!\langle K \rangle\!\rangle$ obtained by merging the configuration can reduce to $\langle\!\langle K' \rangle\!\rangle$, for any store compatible with K. On the other hand, if a process does a reduction step then, according to the lts, it can either perform a request transition or a tau transition, depending on whether (CRed Request) was used in the derivation.

Lemma 5.3.16 (Reductions) *Tau transitions between configurations imply reductions between the corresponding networks. For all D, K such that $\mathit{dom}(K) \subseteq \mathit{dom}(D)$*

1. *if $K \xrightarrow{l\cdot\tau} K'$ then $(D, \langle\!\langle K \rangle\!\rangle) \to (D, \langle\!\langle K' \rangle\!\rangle)$;*
2. *if $K \xrightarrow{\tau} \!\!\!\!\to_\Lambda K'$ then $(D, \langle\!\langle K \rangle\!\rangle) \to^* (D, \langle\!\langle K' \rangle\!\rangle)$.*

Proof. Point 1 follows from point 2 of Lemma 5.3.12. Point 2 follows from point 1 noticing that tau transitions do not increase the domain of a configuration. □

Lemma 5.3.17 (Transitions) *Reductions between networks imply tau or request transitions between the corresponding configurations. If $(D, P) \longrightarrow (D_1, P_1)$ then one of the following holds:*

1. *$P \xrightarrow{l\cdot\tau} P_1$ and $D = \{l \mapsto T\} \uplus E = D_1$;*

2. $P \xrightarrow{(\widetilde{k})l\cdot\mathsf{req}\langle p'\rangle(T')} P_2 \,|\, \Theta^{\widetilde{k}}$ *and* $\begin{cases} P_1 \equiv \langle\!\langle P_2 \,|\, \Theta^{\widetilde{i}} \rangle\!\rangle, \\ D = \{l \mapsto T\} \uplus E, \\ D_1 = \{l \mapsto T_1\} \uplus E, \end{cases}$ *where there exists a path* p *such that* $\begin{cases} \mathfrak{E}(p, T) = (T_1, U_1 \shortmid \ldots \shortmid U_n \shortmid \varnothing), \\ \mathfrak{X}(p) = (p'; \Theta^{\widetilde{k}}), \\ \mathfrak{X}(\mathtt{r}[\,U_1\,] \shortmid \ldots \shortmid \mathtt{r}[\,U_n\,] \shortmid \varnothing) = (T'; \Theta^{\widetilde{i}}). \end{cases}$

Proof. Both points follow by induction on the depth of the derivation tree of $(D, P) \to (D_1, P_1)$, using points 3 and 4 of Lemma 5.3.3 to derive the labelled transition from the structure of the processes as revealed by the reduction step. □

We know by Definition 5.1.1 that a script-independent query language, starting from queries and input trees which are equivalent up-to substitutions of scripts for trigger names, gives equivalent output trees and results. The lemma below relates this notion to extraction.

Lemma 5.3.18 (Extraction) *Consider an arbitrary script-independent query language. Suppose* $\mathfrak{X}(T) = (T_0; \Theta^{\widetilde{k}})$, $\mathfrak{X}(p) = (p'; \Theta^{\widetilde{j}})$, $\mathfrak{E}(p, T) = (T_1, U_1 \shortmid \ldots \shortmid U_n \shortmid \varnothing)$, $\mathfrak{X}(\mathtt{r}[\,U_1\,] \shortmid \ldots \shortmid \mathtt{r}[\,U_n\,] \shortmid \varnothing) = (T'; \Theta^{\widetilde{i}})$ *and* $\mathfrak{X}(T_1) = (T_1'; \Theta^{\widetilde{h}})$.

1. *for any definition* $\langle k \Leftarrow A \rangle$ *occurring in* $\Theta^{\widetilde{i}}$ *or* $\Theta^{\widetilde{h}}$ *there must be a definition* $\langle k' \Leftarrow A \rangle$ *occurring in* $\Theta^{\widetilde{k}}$ *or* $\Theta^{\widetilde{j}}$.

2. *if* $\mathfrak{X}(S) = (T_0; \Omega^{\widetilde{k}})$ *and* $\mathfrak{X}(q) = (p'; \Omega^{\widetilde{j}})$, *then* $\mathfrak{E}(q, S) = (S_1; V_1 \shortmid \ldots \shortmid V_n \shortmid \varnothing)$, $\mathfrak{X}(\mathtt{r}[\,V_1\,] \shortmid \ldots \shortmid \mathtt{r}[\,V_n\,] \shortmid \varnothing) = (T'; \Omega^{\widetilde{i}})$ *and* $\mathfrak{X}(S_1) = (T_1'; \Omega^{\widetilde{h}})$.

Proof. Both points follow easily by Definition 5.1.1 and Observation 5.1.3. □

We need to compare domain bisimilarity, which is defined on configurations without taking the store into account, with domain congruence, which is defined using reduction congruence (a relation on networks). We can do this because bisimilarity requires a correspondence between matching actions of two configurations, which implies that the stores in the networks corresponding to the configurations can diverge, after each reduction step, only up-to equivalent scripts. To formalize this intuition, we introduce the relation $\asymp$ on networks.

Definition 5.3.19 (Candidate Relation) *We define the* candidate relation $\asymp$ *by*

$$\asymp \stackrel{\text{def}}{=} \left\{ ((D, P), (B, Q)) \;\; : \;\; \mathfrak{X}(D) = (D'; \Theta^{\widetilde{k}}),\, \mathfrak{X}(B) = (D'; \Omega^{\widetilde{k}}),\, P \,|\, \Theta^{\widetilde{k}} \approx_\Lambda Q \,|\, \Omega^{\widetilde{k}} \right\}$$

where $\mathit{dom}(B) = \mathit{dom}(D) = \Lambda$ *and* $(\mathit{fn}(P) \cup \mathit{fn}(Q)) \cap \mathcal{Y} = \emptyset$.

We can show now that the candidate relation $\asymp$ is sound with respect to $\simeq_{\mathrm{r}}$, the relation on networks inducing request congruence.

Lemma 5.3.20 *The candidate relation is contained in the reduction congruence induced by request observables:* $\asymp \subseteq \simeq_{\mathrm{r}}$.

Proof. By definition of $\simeq_{\mathrm{r}}$, we need to show that $\asymp$ is (1) observation preserving, (2) contextual, and (3) reduction-closed.

1. Follows from the definition of request observables, the hypothesis $P \mid \Theta^{\widetilde{k}} \approx_\Lambda Q \mid \Omega^{\widetilde{k}}$ and Lemma 5.3.16, noticing that $\Theta^{\widetilde{k}}$ and $\Omega^{\widetilde{k}}$ cannot perform tau or request transitions.

2. Consider a generic reduction context $(E \uplus -, (\nu\, \widetilde{c})(R \mid -))$. By definition of $\mathfrak{X}$, $\mathfrak{X}(E \uplus D) = (E' \uplus D'; \Phi^{\widetilde{j}} \mid \Theta^{\widetilde{k}})$ and $\mathfrak{X}(E \uplus B) = (E' \uplus D'; \Phi^{\widetilde{j}} \mid \Omega^{\widetilde{k}})$. By hypothesis, $P \mid \Theta^{\widetilde{k}} \approx_\Lambda Q \mid \Omega^{\widetilde{k}}$. By Theorem 5.3.15, $(\nu\, \widetilde{c})(R \mid P \mid \Theta^{\widetilde{k}} \mid \Phi^{\widetilde{j}}) \approx_\Lambda (\nu\, \widetilde{c})(R \mid Q \mid \Omega^{\widetilde{k}} \mid \Phi^{\widetilde{j}})$. Since scripts have no private channel names, by structural congruence we conclude that $(\nu\, \widetilde{c})(R \mid P) \mid \Theta^{\widetilde{k}} \mid \Phi^{\widetilde{j}} \approx_\Lambda (\nu\, \widetilde{c})(R \mid Q) \mid \Omega^{\widetilde{k}} \mid \Phi^{\widetilde{j}}$.

3. Suppose $(D, P) \asymp (B, Q)$ and $(D, P) \longrightarrow (D_1, P_1)$. We need to show that $(B, Q) \stackrel{*}{\longrightarrow} (B_1, Q_1) \asymp (D_1, P_1)$. For convenience, we report below what $(D, P) \asymp (B, Q)$ means:

$$\mathfrak{X}(D) = (D'; \Theta^{\widetilde{k}}) \tag{5.1}$$

$$\mathfrak{X}(B) = (D'; \Omega^{\widetilde{k}}) \tag{5.2}$$

$$P \mid \Theta^{\widetilde{k}} \approx_\Lambda Q \mid \Omega^{\widetilde{k}} \tag{5.3}$$

$$\mathit{dom}(B) = \mathit{dom}(D) = \Lambda \tag{5.4}$$

   By Lemma 5.3.17, there are two cases:

   1. $P \xrightarrow{l \cdot \tau} P_1$ and $D = \{l \mapsto T\} \uplus E = D_1$.
      By definition of lts and by equation (5.3) above,

$$\begin{array}{ccc} P \mid \Theta^{\widetilde{k}} & \xrightarrow{\quad l \cdot \tau \quad} & P_1 \mid \Theta^{\widetilde{k}} \\ \approx_\Lambda \big\downarrow & & \big\uparrow \approx_\Lambda \\ Q \mid \Omega^{\widetilde{k}} \xrightarrow{\tau}\!\!\!\!\rightarrow_\Lambda & \xrightarrow{\quad l \cdot \tau \quad} & \xrightarrow{\tau}\!\!\!\!\rightarrow_\Lambda Q_1 \mid \Omega^{\widetilde{k}} \end{array}$$

      By syntactical reasoning, $Q \xrightarrow{l \cdot \tau}\!\!\!\!\rightarrow_\Lambda Q_1$.
      By Lemma 5.3.16, $(B, Q) \rightarrow^* (B, Q_1)$. By (5.1), (5.2) and (5.4) we conclude with

$$\begin{array}{ccc} (D, P) & \longrightarrow & (D, P_1) \\ \asymp \big\downarrow & & \big\uparrow \asymp \\ (B, Q) & \stackrel{*}{\longrightarrow} & (B, Q_1) \end{array}$$

   2. $P \xrightarrow{(\widetilde{j})\, l \cdot \mathsf{req}\langle p' \rangle (T')} P_2 \mid \Theta^{\widetilde{j}}$ where

$$\begin{array}{l} P_1 \equiv \langle\!\langle P_2 \mid \Theta^{\widetilde{i}} \rangle\!\rangle, \\ D = \{l \mapsto T\} \uplus E, \\ D_1 = \{l \mapsto T_1\} \uplus E \end{array}$$

for some p such that

$$\begin{array}{l}\mathfrak{E}(p,T) = (T_1, U_1 \mid ... \mid U_n \mid \varnothing),\\ \mathfrak{X}(p) = (p'; \Theta^{\widetilde{j}}),\\ \mathfrak{X}(\mathtt{r}[\,U_1\,] \mid \ldots \mid \mathtt{r}[\,U_n\,] \mid \varnothing) = (T'; \Theta^{\widetilde{i}}).\end{array}$$

By definition of lts and by (5.3),

$$\begin{array}{ccc} P \mid \Theta^{\widetilde{k}} & \xrightarrow{(\widetilde{j})l\cdot\mathsf{req}\langle p'\rangle(T')} & P_2 \mid \Theta^{\widetilde{j}} \mid \Theta^{\widetilde{k}} \\ \approx_\Lambda \big\downarrow & & \big\uparrow \approx_\Lambda \\ Q \mid \Omega^{\widetilde{k}} \xrightarrow{\tau}\!\!\!\!\rightarrow_\Lambda & \xrightarrow{(\widetilde{j})l\cdot\mathsf{req}\langle p'\rangle(T')} & \xrightarrow{\tau}\!\!\!\!\rightarrow_\Lambda Q_2 \mid \Omega^{\widetilde{j}} \mid \Omega^{\widetilde{k}} \end{array} \tag{5.5}$$

By syntactical reasoning and point 4 of Lemma 5.3.3,

$$Q \xrightarrow{\tau}\!\!\!\!\rightarrow_\Lambda (\nu\, a)(Q_3 \mid l\cdot\mathsf{req}_q\langle c\rangle) = Q_M \tag{5.6}$$

$$Q_M \xrightarrow{(\widetilde{j})l\cdot\mathsf{req}\langle p'\rangle(T')} (\nu\, a)(Q_3 \mid \overline{l\cdot c}\langle T'\rangle) \mid \Omega^{\widetilde{j}} \xrightarrow{\tau}\!\!\!\!\rightarrow_\Lambda Q_2 \mid \Omega^{\widetilde{j}} \tag{5.7}$$

for some q such that $\mathfrak{X}(q) = (p'; \Omega^{\widetilde{j}})$. By (5.6) and Lemma 5.3.16,

$$(B, Q) \longrightarrow^* (B, Q_M).$$

By (5.1) and definition of $\mathfrak{X}$,

$$D' = \{l \mapsto T_0\} \uplus E_0 \ \textit{and}\ \Theta^{\widetilde{k}} = \Theta^{\widetilde{k'}} \mid \Theta^{\widetilde{k''}}$$

where $\mathfrak{X}(\{l \mapsto T\}) = (\{l \mapsto T_0\}; \Theta^{\widetilde{k'}})$ and $\mathfrak{X}(E) = (E_0; \Theta^{\widetilde{k''}})$.
By by (5.2) and a similar argument,

$$B = \{l \mapsto S\} \uplus E' \ \textit{and}\ \Omega^{\widetilde{k}} = \Omega^{\widetilde{k'}} \mid \Omega^{\widetilde{k''}}$$

where $\mathfrak{X}(\{l \mapsto S\}) = (\{l \mapsto T_0\}; \Omega^{\widetilde{k'}})$ and $\mathfrak{X}(E') = (E_0; \Omega^{\widetilde{k''}})$.
Suppose $\mathfrak{X}(T_1) = (T_1'; \Theta^{\widetilde{h}})$. By point 2 of Lemma 5.3.18,

$$\begin{array}{l}\mathfrak{E}(q,S) = (S_1; V_1 \mid ... \mid V_n \mid \varnothing),\\ \mathfrak{X}(\mathtt{r}[\,V_1\,] \mid \ldots \mid \mathtt{r}[\,V_n\,] \mid \varnothing) = (T'; \Omega^{\widetilde{i}}),\\ \mathfrak{X}(S_1) = (T_1'; \Omega^{\widetilde{h}}).\end{array}$$

By definition of reduction,

$$(B, Q_M) \longrightarrow (B_1, (\nu\, a)(Q_3 \mid \overline{l\cdot c}\langle \mathtt{r}[\,V_1\,] \mid \ldots \mid \mathtt{r}[\,V_n\,] \mid \varnothing\rangle))$$

By point 2 of Lemma 5.3.17,

$$(\nu\, a)(Q_3 \mid \overline{l\cdot c}\langle \mathtt{r}[\,V_1\,] \mid \ldots \mid \mathtt{r}[\,V_n\,] \mid \varnothing\rangle) \equiv \langle\!\langle (\nu\, a)(Q_3 \mid \overline{l\cdot c}\langle T'\rangle) \mid \Omega^{\widetilde{i}} \rangle\!\rangle$$

By syntactical reasoning on (5.7),

$$(\nu\, a)(Q_3 \mid \overline{l{\cdot}c}\langle T'\rangle) \xrightarrow{\tau} \!\!\!\rightarrow_\Lambda Q_2.$$

By point 2 of Lemma 5.3.12,

$$\langle\!\langle (\nu\, a)(Q_3 \mid \overline{l{\cdot}c}\langle T'\rangle) \mid \Omega^{\widetilde{i}}\rangle\!\rangle \xrightarrow{\tau} \!\!\!\rightarrow_\Lambda \langle\!\langle Q_2 \mid \Omega^{\widetilde{i}}\rangle\!\rangle.$$

By Lemma 5.3.16,

$$(B_1, \langle\!\langle (\nu\, a)(Q_3 \mid \overline{l{\cdot}c}\langle T'\rangle) \mid \Omega^{\widetilde{i}}\rangle\!\rangle) \longrightarrow^* (B_1, \langle\!\langle Q_2 \mid \Omega^{\widetilde{i}}\rangle\!\rangle)$$

By two applications of point 1 of Lemma 5.3.18, for any definition $\langle i \Leftarrow A\rangle$ occurring in $\Theta^{\widetilde{i}}$ or $\Theta^{\widetilde{h}}$ there must be a definition $\langle k' \Leftarrow A\rangle$ occurring in $\Theta^{\widetilde{k}}$ or $\Theta^{\widetilde{j}}$.
By (5.5) and by Lemma 5.3.13,

$$P_2 \mid \Theta^{\widetilde{h}} \mid \Theta^{\widetilde{k''}} \mid \Theta^{\widetilde{i}} \approx_\Lambda Q_2 \mid \Omega^{\widetilde{h}} \mid \Omega^{\widetilde{k''}} \mid \Omega^{\widetilde{i}}.$$

By using an appropriate instance of the candidate bisimulation in the proof of Lemma 5.3.14,

$$\langle\!\langle P_2 \mid \Theta^{\widetilde{i}}\rangle\!\rangle \mid \Theta^{\widetilde{h}} \mid \Theta^{\widetilde{k''}} \approx_\Lambda \langle\!\langle Q_2 \mid \Omega^{\widetilde{i}}\rangle\!\rangle \mid \Omega^{\widetilde{h}} \mid \Omega^{\widetilde{k''}},$$

and we conclude with

$$\begin{array}{ccc} (D, P) & \longrightarrow & (D_1, \langle\!\langle P_2 \mid \Theta^{\widetilde{i}}\rangle\!\rangle) \\ \asymp \downarrow & & \uparrow \asymp \\ (B, Q) & \longrightarrow^* & (B_1, \langle\!\langle Q_2 \mid \Omega^{\widetilde{i}}\rangle\!\rangle) \end{array}$$

□

We have now all the tools necessary to show the soundness of domain bisimilarity with respect to request congruence.

Theorem 5.3.21 (Soundness) *Domain bisimilarity is a sound approximation of the domain congruence induced by request observables: for all $\Lambda, \boldsymbol{P}, \boldsymbol{Q}$ where $(fn(\boldsymbol{P}) \cup fn(\boldsymbol{Q})) \cap \mathcal{Y} = \emptyset$, if $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q}$ then $\boldsymbol{P} \sim_r^\Lambda \boldsymbol{Q}$.*

Proof. By definition, $\boldsymbol{P} \sim_r^\Lambda \boldsymbol{Q}$ if and only if $(D, C[\boldsymbol{P}]) \simeq_{\mathrm{r}} (D, C[\boldsymbol{Q}])$ for all $D, C[-]$ such that $\Lambda \subseteq \mathit{dom}(D)$ and $C[-]$ does not contain trigger names and is closing for both $\boldsymbol{P}$ and $\boldsymbol{Q}$. Suppose $\boldsymbol{P} \approx_\Lambda \boldsymbol{Q}$ and consider some arbitrary $D, C[-]$ respecting the conditions above. Suppose $\mathfrak{X}(D) = (D'; \Theta^{\widetilde{k}})$. By point 2 of Theorem 5.3.15, $C[\boldsymbol{P}] \mid \Theta^{\widetilde{k}} \approx_\Lambda C[\boldsymbol{Q}] \mid \Theta^{\widetilde{k}}$. By Definition 5.3.19, $(D, C[\boldsymbol{P}]) \asymp (D, C[\boldsymbol{Q}])$. By Lemma 5.3.20, $(D, C[\boldsymbol{P}]) \simeq_{\mathrm{r}} (D, C[\boldsymbol{Q}])$. □

5.3.4 Fixed-point characterization

As noted in Section 5.2, since our definition of domain bisimilarity is non-standard, we need to argue that the largest domain bisimulation exists. Our proof follows the structure of an analogous proof given by Milner for CCS [56]. We begin by defining an operator whose fixed-point (according to the point-wise inclusion ordering) coincides with domain bisimilarity.

Definition 5.3.22 (Operator $\mathcal{F}$) *Let $\mathcal{R}el(\mathcal{W})_{\mathcal{L}} \stackrel{\text{def}}{=} \wp(\mathcal{L}) \rightharpoonup \wp(\mathcal{W} \times \mathcal{W})$ be the domain of relations on configurations indexed by set of locations. We define the operator $\mathcal{F} : \mathcal{R}el(\mathcal{W})_{\mathcal{L}} \rightharpoonup \mathcal{R}el(\mathcal{W})_{\mathcal{L}}$ as*

$$(K, L) \in \mathcal{F}(\mathcal{R})(\Lambda) \quad \textit{if and only if}$$

- *$K \xrightarrow{\alpha_l} K'$ implies:*
  1. *if $l \in \Lambda$ with $\mathit{rel}(\alpha_l, L)$ then $L \stackrel{\alpha_l}{\twoheadrightarrow}_{\Lambda} Q'$ and $(K', L') \in \mathcal{R}(\Lambda)$;*
  2. *if $l \notin \Lambda$ then $(K, L) \in \mathcal{R}(\Lambda \cup \{l\})$;*
- *$L \xrightarrow{\alpha_l} L'$ implies:*
  1. *if $l \in \Lambda$ with $\mathit{rel}(\alpha_l, K)$ then $K \stackrel{\alpha_l}{\twoheadrightarrow}_{\Lambda} K'$ and $(K', L') \in \mathcal{R}(\Lambda)$;*
  2. *if $l \notin \Lambda$ then $(K, L) \in \mathcal{R}(\Lambda \cup \{l\})$.*

Definition 5.3.23 (Order) *We denote by $\sqsubseteq$ the point-wise ordering on $\mathcal{R}el(\mathcal{W})_{\mathcal{L}}$ given by*

$$\mathcal{R} \sqsubseteq \mathcal{R}' \iff \forall \Lambda \in \wp(\mathcal{L}).\ \mathcal{R}(\Lambda) \subseteq \mathcal{R}'(\Lambda).$$

Below we show that the operator $\mathcal{F}$ defined above is monotonic, and that each domain bisimulation is a pre-fixed-point of $\mathcal{F}$. By these two results and by definition of bisimilarity, it follows that bisimilarity is the largest fixed-point of $\mathcal{F}$.

Lemma 5.3.24 (Largest Fixed-Point)

1. *$\mathcal{F}$ is monotonic;*
2. *$\mathcal{R}$ is a domain bisimulation if and only if $\mathcal{R} \sqsubseteq \mathcal{F}(\mathcal{R})$;*
3. *domain bisimilarity is the largest fixed-point of $\mathcal{F}$.*

Proof.

1. We must show that given $\mathcal{R} \sqsubseteq \mathcal{R}'$ we have $\mathcal{F}(\mathcal{R}) \sqsubseteq \mathcal{F}(\mathcal{R}')$. By Definition 5.3.23 this is equivalent to show that given a generic Λ, $\mathcal{F}(\mathcal{R})(\Lambda) \subseteq \mathcal{F}(\mathcal{R}')(\Lambda)$ knowing that for all Λ, $\mathcal{R}(\Lambda) \subseteq \mathcal{R}'(\Lambda)$. This follows from Definition 5.3.22.
2. By Definition 5.3.23, we have to show that for a generic Λ, $\mathcal{R}(\Lambda) \subseteq \mathcal{F}(\mathcal{R})(\Lambda)$, which follows by Definition 5.2.2.

3. By point 2, each domain bisimulation is a pre-fixed-point and each pre-fixed-point is a domain bisimulation. By Definition 5.2.2, $\approx$ is a domain bisimulation, and being the largest, it is also the largest pre-fixed-point of $\mathcal{F}$. By point 1 (monotonicity) we have that $\mathcal{F}(\approx) \sqsubseteq \mathcal{F}(\mathcal{F}(\approx))$, hence $\mathcal{F}(\approx)$ is a pre-fixed-point. Since $\approx$ is the largest, $\mathcal{F}(\approx) \sqsubseteq \approx$, hence $\mathcal{F}(\approx) = \approx$. Since each fixed-point is also a pre-fixed-point, $\approx$ is the largest fixed-point.

□

Chapter 6

Distributed Query Patterns

In this chapter, we apply our formal techniques to reason about examples of distributed query patterns*, which are protocols used by the peers of data integration systems to answer distributed queries more efficiently. In Section 6.1, we show how to represent and combine in Xdπ three particularly representative distributed query patterns, and we show formally that their semantics satisfies an intuitive specification. In Section 6.2, we propose a new pattern exploiting process mobility, and we prove that it is semantically equivalent to a pattern, based only on remote communication, which needs to transfer more data.*

6.1 Chaining, recruiting and referral

In this section, we consider *chaining*, *recruiting* and *referral*, three distributed query patterns studied by Sahuguet *et al.* in [68, 66] and described below. These patterns are interesting because, as will soon be apparent, they are simple yet can express ways of answering requests which are non-trivial, and display different levels of cooperation between the parties involved.

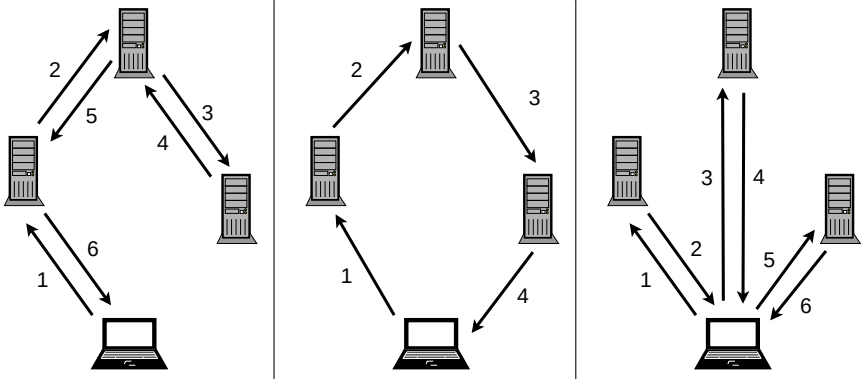
The usage of these patterns presupposes an architecture of servers sharing a common communication protocol for answering cooperatively the queries issued by clients. The protocol consists of alternative actions which depend on the contents of a query and on the local data, and is implemented by dedicated services running on each peer. The distributed querying infrastructure obtained by combining the three query patterns is very flexible and can provide location independence to the clients. In fact, a client simply needs to invoke a service on a peer acting as the "entry point" to the network in order to get access to data which may reside on some other server unknown to the client itself.

We now describe the three patterns. In each case, a server receiving a query will try to execute it locally, and if that is not possible, will take alternative action.

Chaining (Figure 6.1.(A)): if a server cannot deal directly with the call, it reissues it to an alternative server, waits for the answer, and then returns the answer to the client.

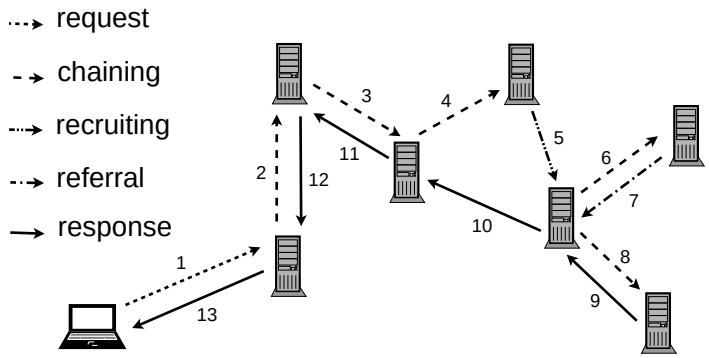
Recruiting (Figure 6.1.(B)): if a server cannot deal directly with the call, it for-

Figure 6.1: Chaining, recruiting and referral



(A) Chaining. (B) Recruiting. (C) Referral.

Figure 6.2: Combining the query patterns



wards it to another server (without notifying the client), so that the result will eventually return to the client without further intervention of the first server. To implement this pattern the address for returning the result must be a parameter of the call, and the client must be willing to accept asynchronous connections.

Referral (Figure 6.1.(C)): if a server cannot deal directly with the call, it suggests to the client an alternative server which might be able to. This strategy requires active collaboration from the client, which must be ready to contact the alternative server.

When each server involved in answering a request is able to use any of the patterns above, the flow of the data from the initial service call to the final answer can become complex and involve arbitrary combinations of the patterns, as in the example shown in Figure 6.2.

6.1.1 Implementing the patterns

We now describe, step by step, some Core X$d\pi$ code which implements a system where a client request can be answered by servers using an arbitrary combination of chaining, recruiting and referral. The code is based on services which retrieve and combine data from different locations by exploiting remote communication and local requests. In Section 3.2, we have seen how to represent service calls in X$d\pi$. The corresponding code in Core X$d\pi$, running on location l, looks like

$$(\nu\, c)(l{\cdot}\overline{m{\cdot}a}\langle\widetilde{v}, n, c\rangle \mid l{\cdot}c(\widetilde{\pi}, y, z).\boldsymbol{P})$$

where a is the name of the service to be invoked at location m with parameters $\widetilde{v}$ yielding a result to be passed on channel c at location l, and $\boldsymbol{P}$ is the code for handling the results, which are expected to match pattern $\widetilde{\pi}$. In this section, a service call will carry four parameters: a tree T used to represent a condition, checked using pattern matching, that a server must satisfy in order to provide the right service (for example specifying the kind of result expected), a query p which is meant to be run on the store of the service matching tag T, and the return parameters n and c stating the location and the channel where the result should be returned. This approach can be easily applied also to service call having more parameters.

A client must be able to deal with the referral query pattern, therefore its code consists essentially in a loop. The loop consists in calling a first server (which could in principle provide the final result, terminating the loop), and then repeating the same call at the alternative addresses received in unsuccessful replies, until a reply containing the final result is received. The context defined below implements the loop at location m:

$$m{\cdot}\mathsf{Ref}_{(n,l,s,T,p,z)}[-] \stackrel{\text{def}}{=} (\nu\, c)\left(\begin{array}{l} m{\cdot}c(\texttt{OK}\,, z).\,- \quad \mid \\ \overline{m{\cdot}c}\langle\texttt{REF}\,, l\rangle \mid !m{\cdot}c(\texttt{REF}\,, x).m{\cdot}\overline{x{\cdot}s}\langle T, p, n, c\rangle \end{array}\right)$$

It is parametric in the location n where the result must be returned, the location l of the first server to be interrogated, and the parameters of the call: the service name s and condition T, the actual query p and the variable z for binding the result in the continuation. The context uses a private channel c to implement the referral loop and uses the tags `OK` and `REF` as guards to exit or continue the loop. Any process built using this context always starts the referral loop by invoking s at l and then waiting for two possible answers: either a referral message with the tag `REF` and the name of an alternative location (bound to x), which starts another iteration of the loop against the corresponding server, or a result message with the tag `OK` and the result of the service call (bound to z), which terminates the loop and passes the result on to the process which replaces the context hole "$-$".

The server filters calls based on the parameter T in order to decide whether they can be served locally or not. Its code consists of the following two processes, run in parallel:

$$l{\cdot}\mathsf{Local}_{(s,T)} \stackrel{\text{def}}{=} !l{\cdot}s(T, x, y, z).(\nu\, c)(l{\cdot}\mathsf{req}_x\langle c\rangle \mid l{\cdot}c(w).l{\cdot}\overline{y{\cdot}z}\langle\texttt{OK}\,, w\rangle)$$

$$l{\cdot}\mathsf{Remote}_{(s,\Delta)} \stackrel{\text{def}}{=} \prod_{(m,S_m)\in\Delta} !l{\cdot}s(S_m, x, y, z).\left(\begin{array}{l} l{\cdot}\overline{m{\cdot}s}\langle S_m, x, y, z\rangle \\ \oplus\, l{\cdot}\overline{y{\cdot}z}\langle\texttt{REF}\,, m\rangle \\ \oplus\, l{\cdot}\mathsf{Ref}_{(l,m,s,S_m,x,w)}[l{\cdot}\overline{y{\cdot}z}\langle\texttt{OK}\,, w\rangle]. \end{array}\right)$$

If the first parameter matches T, the server runs the query (bound to x) on the local data and sends the result back to the client on channel z at y. If the first parameter does not match T, the server selects another server more appropriate for that request out of the set Δ relating servers to tags specifying their services (the outermost parallel composition of the remote process). It then invokes s on the chosen server using either chaining (third branch of the choice), or recruiting (first branch), or referral (second branch). In the case of chaining, the server runs the same code as the client with different parameters. Notice that the code handling the result forwards the result to the client instead of using it locally.

Installation. In order to use these patterns, the code implementing the services must be installed somehow on each participating server. We can assume that it is pre-installed on each peer, or we can install it "on demand" using either process migration or a specialized service which runs scripts. For example, consider the code of a service $\boldsymbol{P}$ parametric in the location x where the service is run and some other initialization pattern π. If we assume that an arbitrary location l exists then, given an arbitrary initialization parameter $v = \pi\sigma$, it is easy to see that running the code at m is equivalent to installing the service code from l

$$\boldsymbol{P}\{^{m}/_{x}\}\sigma \sim_{r}^{\{l\}} l{\cdot}\mathsf{go}\, m.(x,\pi)\boldsymbol{P} \circ \langle m, v\rangle.$$

Alternatively, one could use a dedicated installation service *Inst* at location m which receives an abstraction and some parameters, and runs the abstraction locally

$$\boldsymbol{P}\{^{m}/_{x}\}\sigma \sim_{r}^{\{l\}} (\nu\, \mathit{Inst})(l{\cdot}\overline{m{\cdot}\mathit{Inst}}\langle(x,\pi)\boldsymbol{P}, v\rangle \mid !m{\cdot}\mathit{Inst}(y,z).y \circ \langle m, z\rangle).$$

6.1.2 Relating the patterns to a specification

We use a simple system with a client and two servers as an example of how to reason using our equivalences. The reasoning is analogous in the case of multiple servers. The client is on peer m, and runs the code

$$m{\cdot}\mathsf{Client}_{(l,s,\mathtt{a})} \stackrel{\text{def}}{=} m{\cdot}\mathsf{Ref}_{(m,l,s,\mathtt{a},p,z)}[m{\cdot}\boldsymbol{P}]$$

where the service is requested to match the tag $\mathtt{a}$, and the continuation process $\boldsymbol{P}$ is an arbitrary process located at m which does not contain free occurrences of channels a and c mentioned in the definition of $\mathsf{Ref}_{(-)}$. A server is composed by the parallel composition of the branches dealing with local and remote processing, as described in Section 6.1.1:

$$l_1{\cdot}\mathsf{Server}_{(s,T,l_2,S)} \stackrel{\text{def}}{=} l_1{\cdot}\mathsf{Remote}_{(s,\{(l_2,S)\})} \mid l_1{\cdot}\mathsf{Local}_{(s,T)}.$$

We consider two processes P_1 and P_2, which both request to the server at l_1 some data, but in the first case the data is specified by $\mathtt{a}$ (hence it is served served locally on l_1), whilst in the second case it is specified by $\mathtt{b}$ (hence it is served remotely at l_2)

$$\mathsf{Servers} \stackrel{\text{def}}{=} l_1{\cdot}\mathsf{Server}_{(s,\mathtt{a},l_2,\mathtt{b})} \mid l_2{\cdot}\mathsf{Server}_{(s,\mathtt{b},l_1,\mathtt{a})}$$

$$P_1 \stackrel{\text{def}}{=} m{\cdot}\mathsf{Client}_{(l_1,s,\mathtt{a})} \mid \mathsf{Servers}$$

$$P_2 \stackrel{\text{def}}{=} m{\cdot}\mathsf{Client}_{(l_1,s,\mathtt{b})} \mid \mathsf{Servers}$$

We compare P_1 and P_2 with Q_1 and Q_2 defined below, which provide a specification of the expected behaviour respectively of P_1 and P_2. Each process goes directly from m to the relevant location, fetches the data returned by query p, and goes back to paste it as the new data tree of m:

$$m{\cdot}\mathsf{Spec}_{(l)} \stackrel{\text{def}}{=} m{\cdot}\mathsf{go}\, l.(\nu\, c)(l{\cdot}\mathsf{req}_p\langle c\rangle \mid l{\cdot}c(z).l{\cdot}\mathsf{go}\, m.m{\cdot}\boldsymbol{P})$$

$$Q_1 \stackrel{\text{def}}{=} m{\cdot}\mathsf{Spec}_{(l_1)} \mid \mathsf{Servers}$$

$$Q_2 \stackrel{\text{def}}{=} m{\cdot}\mathsf{Spec}_{(l_2)} \mid \mathsf{Servers}.$$

An important difference between the client and the specification is that the client sends an output message to a service which can in principle be intercepted by some process external to the protocol which performs an input on the same service channel. To rule out this undesired interference, we restrict the name of the service s both in each P_i and Q_i, with the side effect of preventing also the unharmful case in which several clients use the services at the same time[1].

We can show, in a domain containing both l_1 and l_2, the following equivalences:

$$(\nu\, s)P_1 \sim_r^{\{l_1,l_2\}} (\nu\, s)Q_1 \qquad (\nu\, s)P_2 \sim_r^{\{l_1,l_2\}} (\nu\, s)Q_2$$

Hence, by definition, we can replace $(\nu\, s)P_i$ by $(\nu\, s)Q_i$ in any network, and preserve network equivalence.

Proof of equivalence. By virtue of Theorem 5.3.21, a formal proof of each equivalence above involves showing the existence of an appropriate domain bisimulation containing the relevant pair, along the lines of the examples of Section 5.2.

We show the case for P_2 and Q_2. In order to make the proof more manageable, we adopt the simplifying assumption that the query p does not contain scripts. The proof of the general case follows a similar structure. Moreover, we use implicitly the closure of bisimilarity under structural congruence.

We start by analyzing the non-input transitions of the two processes, and then we indicate how to build a domain bisimulation by pairing compatible states and dealing with input transitions. Consider $(\nu\, s)Q_2$. It is easy to see that $\mathsf{Servers}$ does not have transitions, hence we concentrate on $m{\cdot}\mathsf{Spec}_{(l_2)}$. All it can do is a tau transition at l_2 corresponding to a migration step, followed by a request transition at l_2 to become

$$(\nu\, s)((\nu\, c)(\overline{l_2{\cdot}c}\langle V\rangle \mid l_2{\cdot}c(z).l_2{\cdot}\mathsf{go}\, m.m{\cdot}\boldsymbol{P}) \mid \mathsf{Servers})$$

where V stands for a generic result obtained by the request. In turn, this process can only do a local communication followed by a migration (both tau transitions, respectively at l_2 and m) to become

$$S_0 \;=\; (\nu\, s)((\nu\, c)(m{\cdot}\boldsymbol{P}\{^V/_z\}) \mid \mathsf{Servers}).$$

[1] In future, we plan to consider less restrictive ways to rule out this kind of interference using the type based techniques for linearity of Yoshida *et al.* [86].

Using the hypothesis $c \notin \mathit{fn}(\boldsymbol{P})$, we obtain

$$(\nu\, s)((\nu\, c)(m{\cdot}\boldsymbol{P}\{^{V}/_{z}\}) \mid \mathsf{Servers}) \equiv (\nu\, s)(m{\cdot}\boldsymbol{P}\{^{V}/_{z}\} \mid \mathsf{Servers}).$$

We now analyze the transitions of $(\nu\, s)P_2$. First the client $m{\cdot}\mathsf{Client}_{(l_1,s,\mathtt{b})}$ performs a tau transition at m corresponding to the initialization of the loop and then one at l_1 corresponding to the migration of the service call. The whole process becomes

$$(\nu\, s, c)(C_0 \mid \overline{l_1{\cdot}s}\langle \mathtt{b}, p, m, c\rangle \mid \mathsf{Servers})$$

where

$$C_0 = m{\cdot}s(\mathtt{OK}, z).m{\cdot}\boldsymbol{P} \mid !m{\cdot}s(\mathtt{REF}, x).m{\cdot}\overline{x{\cdot}s}\langle \mathtt{b}, p, n, c\rangle.$$

Because of $\mathtt{b}$, $l_1{\cdot}\mathsf{Server}_{(s,\mathtt{a},l_2,\mathtt{b})}$ receives the call in the remote branch (the one for l_2 with tag $\mathtt{b}$). Nondeterministically, the process at l_1 evolves to $l_1{\cdot}\mathsf{Server}_{(s,\mathtt{a},l_2,\mathtt{b})}$ with in parallel either $l_1{\cdot}\overline{l_2{\cdot}s}\langle \mathtt{b}, p, m, c\rangle$ (recruiting), or $l_1{\cdot}\overline{m{\cdot}c}\langle \mathtt{REF}, l_2\rangle$ (referral), or $l_1{\cdot}\mathsf{Ref}_{(l_1,l_2,s,\mathtt{b},p,w)}[l_1{\cdot}\overline{m{\cdot}c}\langle \mathtt{OK}, w\rangle]$ (chaining). Both the communication for receiving the call and the choice of the branch to execute are two tau transitions at l_1. We now consider the transitions of each choice branch.

Recruiting. If recruiting is chosen, the server at l_1 performs a tau transition at l_2 corresponding to the forwarding of the client request, becoming

$$(\nu\, s, c)(C_0 \mid R_1 \mid l_1{\cdot}\mathsf{Server}_{(s,\mathtt{a},l_2,\mathtt{b})} \mid \overline{l_2{\cdot}s}\langle \mathtt{b}, p, m, c\rangle \mid l_2{\cdot}\mathsf{Server}_{(s,\mathtt{b},l_1,\mathtt{a})})$$

where R_1 is a deadlocked process containing the code for the two discarded choice branches. Due to the parameter $\mathtt{b}$, server l_2 receives the call in the local branch, performing another tau transition at l_2:

$$(\nu\, s, c)(\ldots \mid l_2{\cdot}\mathsf{Server}_{(s,\mathtt{b},l_1,\mathtt{a})} \mid (\nu\, c')(l_2{\cdot}\mathsf{req}_p\langle c'\rangle \mid l{\cdot}c'(w).l_2{\cdot}\overline{m{\cdot}c}\langle \mathtt{OK}, w\rangle)).$$

This process can only do a request transition, followed by a tau transition (corresponding to local communication) at l_2, becoming

$$(\nu\, s, c)(C_0 \mid R_1 \mid \mathsf{Servers} \mid l_2{\cdot}\overline{m{\cdot}c}\langle \mathtt{OK}, V\rangle),$$

where V stands for a generic result obtained by the request. After two tau transitions at m, corresponding to migration and local communication between C_0 and $\overline{m{\cdot}c}\langle \mathtt{OK}, V\rangle$, we obtain

$$S_1 \;=\; (\nu\, s)(m{\cdot}\boldsymbol{P}\{^{V}/_{x}\} \mid (\nu\, c)(!m{\cdot}s(\mathtt{REF}, x).m{\cdot}\overline{x{\cdot}s}\langle \mathtt{b}, p, n, c\rangle) \mid R_1 \mid \mathsf{Servers}),$$

where we stop.

Referral. In the case of referral, the server performs two tau transitions at m which correspond to the forwarding of the message referring location l_2, and to a second iteration of the referral loop of the client. The client then sends a new call to l_2 (a tau transition at l_2) and becomes

$$(\nu\, s, c)(C_0 \mid R_2 \mid l_1{\cdot}\mathsf{Server}_{(s,\mathtt{a},l_2,\mathtt{b})} \mid \overline{l_2{\cdot}s}\langle \mathtt{b}, p, m, c\rangle \mid l_2{\cdot}\mathsf{Server}_{(s,\mathtt{b},l_1,\mathtt{a})})$$

where R_2 is a deadlocked process containing the code for the two discarded choice branches. From now on, the transitions are the same as in the case for recruiting until we obtain the process

$$S_2 \;=\; (\nu\, s)(m{\cdot}\boldsymbol{P}\{{}^{V}/_{x}\} \mid (\nu\, c)(!m{\cdot}s(\mathtt{REF}\,, x).m{\cdot}\overline{x{\cdot}s}\langle \mathtt{b}\,, p, n, c\rangle) \mid R_2 \mid \mathsf{Servers}),$$

where we stop.

Chaining. In the case of chaining, the reasoning is similar. The first transitions correspond to a local communication at l_1 starting the referral loop of the server and a migration followed by communication at l_2 to start the local branch of that service. At this point, the process performs a request transition analogous to the one in the previous cases, and tau transitions corresponding to a local communication to get the result at l_2, a migration to l_1, and a local communication to terminate the referral loop of the server. The continuation process performs the transitions corresponding to the migration of the final result to m and to the communication to terminate the referral loop of the client, reaching

$$S_3 \;=\; (\nu\, s)(m{\cdot}\boldsymbol{P}\{{}^{V}/_{x}\} \mid (\nu\, c)(!m{\cdot}s(\mathtt{REF}\,, x).m{\cdot}\overline{x{\cdot}s}\langle \mathtt{b}\,, p, n, c\rangle) \mid R_3 \mid \mathsf{Servers}),$$

where R_3 is a deadlocked process containing the code for the two discarded choice branches and the residual of the referral loop at l_1.

Intuitively, S_0, S_1, S_2 and S_3 are all equivalent states, because they are structurally equivalent to a process of the form

$$(\nu\, s)(m{\cdot}\boldsymbol{P}\{{}^{V}/_{z}\} \mid \mathsf{Servers} \mid \delta).$$

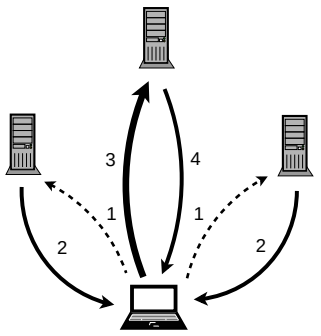
The bisimulation relation we are looking for is obtained in three steps. First, we pair each of the states preceding the request transition in the lts of $(\nu\, s)Q_2$ with each one preceding a request transition in the lts of $(\nu\, s)P_2$ (and vice versa). Second, we pair each of the states following the request transition giving a particular result V in the lts of $(\nu\, s)Q_2$ with each one following a request transition giving the same result in the lts of $(\nu\, s)P_2$ (and vice versa). Third, we close the relation obtained so far under parallel composition with the output messages derived by input transitions (as shown explicitly in Example 5.2.4). It is easy to verify that the relation defined above is a domain bisimulation, by checking the definition.

6.2 Rendez-vous and shipping

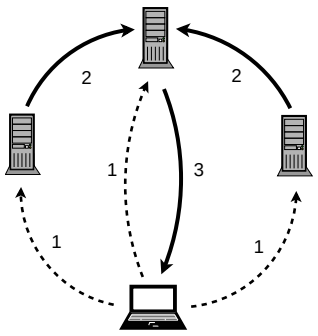
In the previous example, the infrastructure of servers implementing the distributed query patterns was fixed in advance, while the actual interactions between them were determined at run-time. The messages exchanged between different locations were always service calls or their results. Now, we consider a more flexible scenario which exploits code mobility.

Data-shipping and *query-shipping* are two traditional database techniques for distributed query evaluation: the first consists of evaluating locally a query on remote data by asking for the relevant data to be sent from the remote sources; the second

Figure 6.3: Rendez-vous



(A) Query-shipping.



(B) Rendez-vous.

consists of delegating the evaluation of a query to one of the remote sources in order to reduce the bandwidth used by data transfers. In the next section, we propose a distributed query pattern, called *rendez-vous*, which combines data and query shipping by using code mobility and private channels. The idea is to give a client the ability to ship result-handling code to another location, and to redirect the results of arbitrary service calls towards the location containing the result-handler. Within an infrastructure of services such as the one used above for chaining, recruiting and referral, this pattern can help to save bandwidth by eliminating unnecessary data transfers.

6.2.1 The rendez-vous query pattern

We now compare the query-shipping and rendez-vous patterns by giving a concrete example where a client calls a remote service using as parameters two large sets of data obtained by other remote service calls.

Suppose that on location l there is a specialized service $l{\cdot}\texttt{Join}(x_1, x_2, y, z)$ which returns on channel z at location y the result of joining the data bound to x_1 with the data bound to x_2. Suppose moreover that a client running on location m wants to join some data obtained by query p at location l_1 with other data obtained by query q at location l_2. We assume that l_1 and l_2 run the services described in Section 6.1.2, that l_1 (respectively l_2) serves locally the requests tagged by `a` (respectively `b`).

Query shipping. The client can use query shipping: it first invokes the query services at locations l_1 and l_2, then passes on the results as inputs to the join service on location l (see Figure 6.3(A)). Below we give the code of a client implementing this approach:

$$m{\cdot}\mathsf{ClientQ} \stackrel{\text{def}}{=} (\nu\, c, c_1, c_2) \left(\begin{array}{l} m{\cdot}\overline{l_1{\cdot}\texttt{s}}\langle \texttt{a}, p, m, c_1\rangle \\ \mid m{\cdot}\overline{l_2{\cdot}\texttt{s}}\langle \texttt{b}, q, m, c_2\rangle \\ \mid m{\cdot}c_1(\texttt{OK}, x_1).m{\cdot}c_2(\texttt{OK}, x_2).m{\cdot}\overline{l{\cdot}\texttt{Join}}\langle x_1, x_2, m, c\rangle \\ \mid m{\cdot}c(z).m{\cdot}\boldsymbol{P} \end{array} \right)$$

It starts sending off the two service calls to l_1 and l_2 and then waits for the results respectively on c_1 and c_2 to bind them to x_1 and x_2. The remaining code is a standard

service call for the join service at l with parameters x_1 and x_2, binding the final result to z in the continuation $m{\cdot}\boldsymbol{P}$, which can be an arbitrary process.

Rendez-vous. In order to save bandwidth, a better strategy is to request the query services at l_1 and l_2 to forward their results to location l, and to install at l a process which collects the two results and invokes the join service locally, asking for the final result to be returned at location m (see Figure 6.3(B)). Below we give a context implementing the general pattern, with two holes for inserting the code to handle the intermediate results at l and the final result at m. The code is parametric in the tags T_i and the queries p_i used to determine the partial results, the variables x_i for binding them in the intermediate code at "$-_1$", and the variable z for binding the final result in the continuation code at "$-_2$":

$$m{\cdot}\mathsf{RzV}_{(T_1,p_1,x_1,T_2,p_2,x_2,z)}[-]_1[-]_2 \stackrel{\text{def}}{=} (\nu\, c, c_1, c_2)\left(\begin{array}{l} m{\cdot}\overline{l_1{\cdot}\mathtt{s}}\langle T_1, p_1, l, c_1\rangle \\ \mid m{\cdot}\overline{l_2{\cdot}\mathtt{s}}\langle T_2, p_2, l, c_2\rangle \\ \mid m{\cdot}\mathsf{go}\, l.l{\cdot}c_1(\mathtt{OK}, x_1).l{\cdot}c_2(\mathtt{OK}, x_2).-_1 \\ \mid m{\cdot}c(z).-_2 \end{array}\right)$$

The code given above can be easily parameterized also on the number, the names and the locations of the services involved, and can be adapted to return the final results at an arbitrary location on an arbitrary channel.

We give below the code for a client, equivalent to ClientQ, which uses the rendez-vous strategy:

$$m{\cdot}\mathsf{ClientR} \stackrel{\text{def}}{=} m{\cdot}\mathsf{RzV}_{(\mathtt{a},p,x_1,\mathtt{b},q,x_2,z)}[\overline{l{\cdot}\mathtt{Join}}\langle x_1, x_2, m, c\rangle][m{\cdot}\boldsymbol{P}]$$

The code for handling the intermediate results consists in a local call to the join service, whereas the continuation is the same generic process used for ClientQ.

6.2.2 Equivalence of the patterns

Consider the process Servers defined in Section 6.1 consisting in the parallel compositions of the servers for implementing chaining, recruiting and referral at locations l_1 and l_2. The clients given above, each in parallel with Servers, are equivalent in any network regardless of what locations are present.

$$(\nu\, s)(m{\cdot}\mathsf{ClientQ} \mid \mathsf{Servers}) \sim_r^{\emptyset} (\nu\, s)(m{\cdot}\mathsf{ClientR} \mid \mathsf{Servers})$$

In order to make the proof more manageable, we adopt once again the simplifying assumption that p and q do not contain scripts, and we use implicitly the closure of bisimilarity under structural congruence.

First of all, we simplify the problem further by studying an equation relating only the parts of the client processes above which are different from each other and which play a significant role in the proof. The full result follows by exploiting the closure of bisimilarity under parallel composition, restriction and structural congruence (Theorem 5.3.15 and Proposition 5.3.1) to recover the processes of the original statement.

Consider the definitions

$$m{\cdot}\mathsf{ClientQ}' \stackrel{\text{def}}{=} (\nu\, c_1, c_2)\left(\begin{array}{l} m{\cdot}\overline{l_1{\cdot}\mathtt{s}}\langle \mathtt{a}\,, p, m, c_1\rangle \\ \mid\ m{\cdot}\overline{l_2{\cdot}\mathtt{s}}\langle \mathtt{b}\,, q, m, c_2\rangle \\ \mid\ m{\cdot}c_1(\mathtt{OK}\,, x_1).m{\cdot}c_2(\mathtt{OK}\,, x_2).m{\cdot}\overline{l{\cdot}\mathtt{Join}}\langle x_1, x_2, m, c\rangle \end{array}\right)$$

$$m{\cdot}\mathsf{ClientR}' \stackrel{\text{def}}{=} (\nu\, c_1, c_2)\left(\begin{array}{l} m{\cdot}\overline{l_1{\cdot}\mathtt{s}}\langle \mathtt{a}\,, p, l, c_1\rangle \\ \mid\ m{\cdot}\overline{l_2{\cdot}\mathtt{s}}\langle \mathtt{b}\,, q, l, c_2\rangle \\ \mid\ m{\cdot}\mathsf{go}\, l.l{\cdot}c_1(\mathtt{OK}\,, x_1).l{\cdot}c_2(\mathtt{OK}\,, x_2).\overline{l{\cdot}\mathtt{Join}}\langle x_1, x_2, m, c\rangle \end{array}\right)$$

Our goal is to build a domain bisimulation containing the pair

$$\big(\,(\nu\, s)(m{\cdot}\mathsf{ClientQ}' \mid \mathsf{Servers})\,,\ (\nu\, s)(m{\cdot}\mathsf{ClientR}' \mid \mathsf{Servers})\,\big).$$

The construction of the proof is summarized by the diagrams in Figure 6.4. We represent the transitions of the two processes above in the form of lattices of states related by the lts (to be read in the direction of the arrows). Like in Section 6.1.2, we do not consider input transitions at this stage. The dotted arcs indicate the states from the two diagrams which will be paired in the bisimulation.

Building the transition diagrams. We describe the steps leading to the transitions. Later, we will explain how to build the bisimulation relation.

We begin with $(\nu\, s)(m{\cdot}\mathsf{ClientQ}' \mid \mathsf{Servers})$, corresponding to the top diagram of Figure 6.4. The starting state is the one pointed to by an arrow on the left of the diagram. We follow the top-left border of the diagram. Consider the sub-processes $m{\cdot}\overline{l_1{\cdot}\mathtt{s}}\langle \mathtt{a}\,, p, m, c_1\rangle$ and

$$l_1{\cdot}\mathsf{Local}_{(s,\mathtt{a})} \stackrel{\text{def}}{=} !l_1{\cdot}s(\mathtt{a}\,, x, y, z).(\nu\, c')(l_1{\cdot}\mathsf{req}_x\langle c'\rangle \mid l_1{\cdot}c'(w).l_1{\cdot}\overline{y{\cdot}z}\langle \mathtt{OK}\,, w\rangle).$$

Together they can perform, in order:

1. a migration step from m to l_1;
2. an internal communication on s at l_1;
3. a request transition generating an output $\overline{l_1{\cdot}c'}\langle V_1\rangle$, where V_1 is the data obtained by query p;
4. an internal communication on c';
5. a migration to m.

We are left with the processes $l_1{\cdot}\mathsf{Local}_{(s,\mathtt{a})}$ and $\overline{m{\cdot}c_1}\langle \mathtt{0k}\,, V_1\rangle$. The second process can communicate with

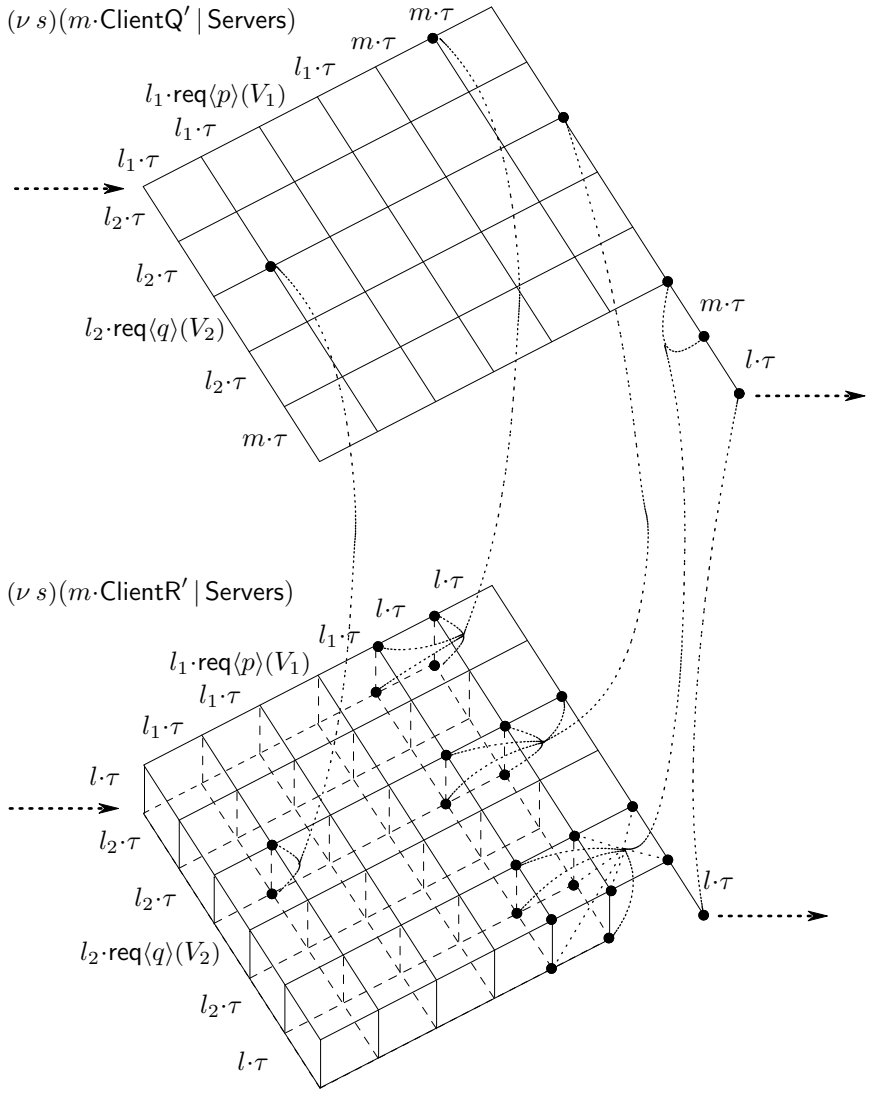
$$m{\cdot}c_1(\mathtt{OK}\,, x_1).m{\cdot}c_2(\mathtt{OK}\,, x_2).m{\cdot}\overline{l{\cdot}\mathtt{Join}}\langle x_1, x_2, m, c\rangle,$$

and we are left with

$$m{\cdot}c_2(\mathtt{OK}\,, x_2).m{\cdot}\overline{l{\cdot}\mathtt{Join}}\langle V_1, x_2, m, c\rangle.$$

Independently, $m{\cdot}\overline{l_2{\cdot}\mathtt{s}}\langle \mathtt{b}\,, q, m, c_2\rangle$ and $l_2{\cdot}\mathsf{Local}_{(s,\mathtt{b})}$ can mimic the 5 steps above (represented in the diagram by the bottom-left border), becoming $l_2{\cdot}\mathsf{Local}_{(s,\mathtt{b})}$ and $\overline{m{\cdot}c_2}\langle \mathtt{0k}\,, V_2\rangle$.

Figure 6.4: Bisimulation Diagrams



These two independent groups of respectively 6 and 5 ordered transitions give a lattice of 42 processes related by the lts, where the bottom element is the initial process, and the top element (at the intersection between the top and bottom-right borders) is the process

$$(\nu\, s)\left(\begin{array}{l} \overline{m{\cdot}c_2}\langle \mathtt{Ok}\,, V_2\rangle \\ \mid\ m{\cdot}c_2(\mathtt{OK}\,, x_2).m{\cdot}\overline{l{\cdot}\mathtt{Join}}\langle V_1, x_2, m, c\rangle \\ \mid\ \mathsf{Servers} \end{array}\right)$$

This process can only perform a communication on channel c_2 and a migration from m to l, becoming the point on the right with the outgoing arrow

$$(\nu\, s)(\overline{l{\cdot}\mathtt{Join}}\langle V_1, V_2, m, c\rangle \mid \mathsf{Servers})$$

By a similar reasoning, we can derive for the process $(\nu\, s)(m{\cdot}\mathsf{ClientQ}' \mid \mathsf{Servers})$ the lattice of $36 + 42 + 1$ processes reported in the bottom diagram of Figure 6.4. The vertical transition possible from each of the 36 states in the lower layer of the diagram is the initial migration step from m to l of the result handling code, where

$$m{\cdot}\mathsf{go}\, l.l{\cdot}c_1(\mathtt{OK}\,, x_1).l{\cdot}c_2(\mathtt{OK}\,, x_2).\overline{l{\cdot}\mathtt{Join}}\langle x_1, x_2, m, c\rangle$$

becomes process

$$l{\cdot}c_1(\mathtt{OK}\,, x_1).l{\cdot}c_2(\mathtt{OK}\,, x_2).\overline{l{\cdot}\mathtt{Join}}\langle x_1, x_2, m, c\rangle.$$

Only when this transition has occurred, can communication on c_1 at l happen (hence the additional 7 states appearing only in the upper layer). The final state, after communication on c_2 at l has happened, is once again

$$(\nu\, s)(\overline{l{\cdot}\mathtt{Join}}\langle V_1, V_2, m, c\rangle \mid \mathsf{Servers}).$$

Building the bisimulation relation. We now describe how to pair-up the processes (states) of Figure 6.4 to build a suitable bisimulation. First, we relate each of the 25 processes in the top diagram reachable from the initial one after at most 4 transitions along each axis, with the two corresponding processes in the bottom diagram (as shown by the left-most dotted arc in Figure 6.4). Then, we relate each of the 10 processes obtained in the top diagram after 5 transitions along one axis and at most 4 on the other axis with the corresponding 4 processes in the bottom diagram (as shown by the second dotted arc in Figure 6.4). Next, we relate each of the 5 processes obtained in the top diagram after 6 transitions along the first axis and at most 4 along the second axis, with the corresponding 5 processes in the bottom diagram (as shown by the third dotted arc in Figure 6.4). Then, we relate the two processes reachable after 11 and 12 transitions in the top diagram with the 10 processes of the bottom diagram to which they are joined by the fourth arc in Figure 6.4. We also relate the process obtained in the top diagram after 5 transitions along each axis with the 8 processes at the vertices of the corresponding cube in the bottom diagram. Finally, we associate the final process in the top diagram with the final process in the bottom diagram (the fifth dotted arc in Figure 6.4).

We take the symmetric closure of this relation, and we close it under parallel composition with output messages like in Example 5.2.4. Note that in the diagram we have shown the transitions for one possible choice of the data items V_1 and V_2 obtained as results of the request transition. To be completely formal, the reasoning above must be quantified on all possible result values, by pairing the corresponding states as in the example of Section 6.1.2.

Following a simulation step. We consider now an example to explain the rationale behind the pairing of states in the relation. We focus on the simulation of $(\nu\, s)(m{\cdot}\mathsf{ClientQ}' \mid \mathsf{Servers})$ by $(\nu\, s)(m{\cdot}\mathsf{ClientR}' \mid \mathsf{Servers})$ which is subtle. The other direction is straightforward.

In the top diagram, the process connected to the leftmost dotted arc can perform a weak transition $\xrightarrow{l_1{\cdot}\mathsf{req}\langle p\rangle(V_1)}\!\!\!\!\rightarrow_{\{l_1,m\}}$ to become the process connected to the third dotted arc. In the bottom diagram, the top process connected to the arc can simulate the step by performing a weak transition $\xrightarrow{l_1{\cdot}\mathsf{req}\langle p\rangle(V_1)}\!\!\!\!\rightarrow_{\{l_1\}}$ to become the

top-left process of those connected to the bottom end of the third arc. The bottom process connected to the first arc cannot simulate the transition by reaching the same process, because that would involve using location l. Instead, it performs a transition $\xrightarrow{l_1 \cdot \mathsf{req}\langle p \rangle (V_1)}\!\!\!\!\rightarrow_{\{l_1\}}$ to become the bottom-left process of those connected to the bottom end of the third arc, which is also in the relation. Also the states reached by the process on the top diagram by performing one or two transitions the less are related to the two states of the bottom diagram mentioned above. The idea is that the last two tau transitions at m cannot be matched by tau transitions at l, in fact they do not need to be matched at all, so the processes in the bottom diagram stay the same.

The association between the states of the two diagrams above is not completely straightforward because we are showing the most general result in which domain congruence holds for the empty domain ($\sim_r^{\emptyset}$). In this case, we had to make sure that for each transition between processes in the top diagram there was a corresponding transition in the second diagram (possibly null) which related two processes bisimilar to the original ones *involving only locations used also by the original transition.* The problems are due to the additional transitions at l which are possible in the second diagram. If we tried to prove $\sim_r^{\{l\}}$ instead, the association between processes would have been straightforward.

Chapter 7

Concluding Remarks

In this chapter, we draw our conclusions. In Section 7.1, we review the contents of the thesis, discussing the merits and the shortcomings of our approach, and the previous work on Xdπ. In Section 7.2, we propose directions for future work on Xdπ and dynamic Web data.

7.1 Review

Our work shows that a behavioural understanding of systems for the exchange of dynamic data on the Web can be grounded on the existing research on process calculi, and requires new reasoning techniques as well as well-established ones.

The X$d\pi$ model. Chapter 2 introduced X$d\pi$, a simple calculus for describing the interaction between data and processes across distributed locations. We used a simple data model consisting of unordered labelled trees, with embedded processes and links to other parts of the network, and processes based on the π-calculus. We tried to keep the definition of X$d\pi$ to a minimum, by including only the simple operations of asynchronous local communication based on pattern matching, execution of a query-update expression on the local repository, migration, spawning of scripted code and creation of fresh channels. As shown in Chapter 3, these basic operations were sufficient to derive conditional statements, nondeterministic choices, constructs for parsing and iterating on list-like structures and remote communication in the style of Web services. We have used these derived constructs to give a precise semantics to AXML-like behaviour, and to extend its expressivity by generalizing service calls to arbitrary scripts and exploiting code mobility. Overall we have found that the use of many derived constructs makes the representation of the examples more readable, at the expense of losing the direct correspondence to the operational semantics.

An important design choice, enabling us to study how properties of data can be affected by process interaction, was to model data and processes at the same level of abstraction, rather than encoding data into processes, as customary in the π-calculus [56, 57, 58, 71]. While such an encoding can make sense when using the π-calculus as a low level concurrency modelling language, it becomes a burden when reasoning about coordination of higher-level processes. Our choice gave us also the opportunity to keep our language modular with respect to the choice of a query

language, which can be easily adapted from the existing literature on XML [12].

With hindsight, we can now identify two design choices worth reconsidering. First, pointers need not be primitive objects. A pointer $p@l$ could be simply represented by a script receiving a private channel x and sending the pair p, l on x, as for example in

$$[\![\texttt{link}[\, p@l\,]]\!] = \texttt{link}[\, \langle (x)\overline{x}\langle p, l\rangle\rangle\,].$$

Second, rather than delegating migration control to external security checks, it would be interesting to include at the location level a construct to constrain process migration based on types, along the lines of [40]. We have indeed already considered such an extension, which has not been included in this thesis because it is of limited interest in the untyped setting. A refined t treatment of migration could also lead to consider interesting failure scenarios [29].

Equivalences. In Chapter 4, we investigated behavioural equivalences for X$d\pi$. Our approach does not focus on the communication actions of processes, which are the basis of observational equivalences for process calculi. Instead, it uses as the basic observable property what data can eventually be present at a given location, which is of central concern in reasoning about dynamic Web data, and compares the equivalences induced by these and other observables. Network equivalences are parametric with respect to the language used for querying and updating documents, and can be instantiated to specific cases.

In order to define process equivalences in terms of network equivalences, and to reason about groups of processes interacting across several locations, we have found it convenient to translate X$d\pi$ into Core X$d\pi$, which is a semantically equivalent calculus where processes are located explicitly and are separated from the data store. In Core X$d\pi$, it is easy to express a partial specification of a network by means of located processes running in parallel, possibly sharing private names. Migration has been included in Core X$d\pi$ only to maintain a closer correspondence with X$d\pi$, but is not necessary. In fact, each located action already contains information about where it is to be executed. For example, in the process below we can imagine that each input on a_i at l_i is followed by an implicit migration step $l_i{\cdot}\mathsf{go}\ \ l_{i+1}$ to the next location:

$$l_1{\cdot}a_1(x).l_2{\cdot}a_2(y).\overline{l_3{\cdot}a_3}\langle x, y\rangle,$$

intuitively corresponds to

$$l_1{\cdot}a_1(x).l_1{\cdot}\mathsf{go}\ \ l_2.l_2{\cdot}a_2(y).l_2{\cdot}\mathsf{go}\ \ l_3.\overline{l_3{\cdot}a_3}\langle x, y\rangle.$$

Overall, if X$d\pi$ were to be discarded completely, in favour of using the fragment of Core X$d\pi$ without migration (which is more fundamental and is at the right level where to carry on equational reasoning) the presentation of our model would be to some extent simpler. Nevertheless, we have decided to remain with our original presentation because we think that is more intuitive, and hopefully helps the reader less familiar with process algebras to understand the examples. Moreover, the encoding of X$d\pi$ in Core X$d\pi$ and its full abstraction (Theorem 4.3.15) are interesting results in their own right, constituting the first formal proof of full abstraction of between a distributed π-calculus and a π-calculus without explicit locations, confirming the informal thesis of [17] that locations can be encoded without divergence in ${}^{e}\pi$.

Process equivalences are typically difficult to use directly because they require a costly property of closure under contexts. To overcome this problem, in Chapter 5 we defined *domain bisimilarity*, a coinductive equivalence relation which entails process equivalence. The definition of domain bisimilarity is non-standard, due to the fact that scripts (which can appear in data) are part of the values, and process equivalences are sensitive to the set of locations constituting the network. Domain bisimilarity generalizes the notion of bisimulation to families of relations indexed by sets of locations, and is based on a labelled transition system which incorporates ideas on asynchronous transitions from [44], and on translating higher-order actions into first order actions from [69, 46].

Domain bisimilarity is intrinsically incomplete, due to its being parametric on a query and update language. In fact, without specializing the labelled transition system to a particular language, we are forced to distinguish request transitions as soon as they contain queries which are syntactically different. On the other hand, equivalences dependent on specific knowledge of the equational theory of queries would lead to optimizations which are no longer correct when the query language changes. Even if, for the sake of argument, we fixed a concrete query-update language and knew everything about its semantic equivalences, it would still be unclear how to deal with the case of Example 5.2.5. There, the initial updates that two processes can perform are by no means equivalent, yet by an "idempotence" argument the overall behaviours turn out to be equivalent. A complete bisimilarity would need to be able to consider is some way the cumulative effect of request transitions, also when interleaved with communication steps, in order to equate sequences of updates with the same global effect on the data-store.

Applications. Using bisimilarity, in Chapter 6 we studied some communication patterns employed by servers in distributed query systems to answer queries from clients. Queries took the form of processes which retrieve and combine data from different locations by using remote communication and local requests. In particular, we considered chaining, recruiting and referral, three distributed query patterns studied in [68, 66] which are interesting because, despite their simplicity, they can express ways of answering requests which are non-trivial and display different levels of cooperation between the parties involved. By exploiting process migration, we have proposed the rendez-vous query pattern which can help to save bandwidth in certain applications.

A challenging application which for reasons of complexity has eluded us, and which we leave to future work, is to extend the distributed query pattern examples to model a robust system where the servers return streams of results which can dry out or be restarted, and show its equivalence to a simpler, non-streaming specification. We believe that our techniques are suitable for this task, but we think that there is a need for automated tools and symbolic techniques (such as the *open bisimulation* of Sangiorgi [70]), which help in producing manageable bisimilarity proofs. For example, even in the simple example given in Section 6.2.2, the bisimulation relation was difficult to represent succinctly because each state of one process could be related to several states of the other process.

7.1.1 Publication history

The ideas in this dissertation evolve from those presented in previous publications on X$d\pi$ and Core X$d\pi$. Overall, the most significant improvements of the presentation of our model over the ones described further below are:

- the simplification of the command for interacting with the store, which used to bound the results of a query to variables in the continuation process, and now returns the results as an output message;
- the independence of the whole framework from the choice of a particular query language, which in precedence was fixed;
- the introduction of binding patterns instead of simple variables to parse input messages.

Minor changes include syntactical differences, the change from an unordered data model to an ordered one, and the parametrization of scripts (which in previous presentations were closed processes) by bound variables, which can be bound to run-time parameters passed to the script upon invocation.

We now briefly overview our previous work on dynamic Web data.

1. *Modelling Dynamic Web Data*, with Philippa Gardner. The workshop paper [32] contains the first definition of X$d\pi$, which has evolved significantly since then. The technical report [33] introduces an "ancestor" of Core X$d\pi$ called Xπ_2, and proposes a proof technique for process equivalence based on higher-order bisimulation for process languages, as studied for example in [75, 25, 69]. The journal version [34] revises slightly the first definition of X$d\pi$, and presents several examples of dynamic Web data. It introduces Core X$d\pi$ and formalizes its equivalence to X$d\pi$, but it does not contain a proof method for process equivalence.

2. *Behavioural Equivalences for Dynamic Web Data*, with Philippa Gardner [55]. This paper contains a previous formulation of Core X$d\pi$ (presented as a standalone calculus), and the first definition of domain bisimilarity. Domain bisimilarity improves on the higher-order bisimulation of [33] by equating more terms, while remaining a sound approximation of process equivalence. The proof of soundness is similar to the one presented here in Chapter 5, but the lts is different (due to the different definition of Core X$d\pi$).

3. *Dynamic Net Data: Theory and Experiment*, with Philippa Gardner, Nobuko Yoshida and Alex Ahern [35]. This poster outlines the X$d\pi$ framework and delineates some related and future work on implementation and security (also mentioned here in Section 7.2), which is part of a project on dynamic Web data funded by an EPSRC e-Science grant.

4. *Process calculi and peer-to-peer Web data integration* [53]. This short paper contains an extract from Chapter 1 of this thesis.

7.2 Future work

We now delineate directions for future work on X$d\pi$ and dynamic Web data.

Types and security. Type systems for X$d\pi$ would be useful to guarantee the absence of run-time errors, refine the behavioural equivalences, guarantee the conformance of data trees to schemas, and study security properties.

We have already begun to investigate a basic type system to guarantee safety, intended as the absence of run-time errors due to objects of a specific sort being found where objects of another sort are expected. We plan to extend the basic system with access-control capabilities, to guarantee that well-typed terms do not violate security assumptions. Our preliminary studies were based on the techniques of Pierce and Sangiorgi [63] (for the π-calculus), and Hennessy and Riely [42] (for $d\pi$), but involved also a substantial reworking due to the presence of data storage and scripted processes. The treatment of location types was also influenced by our work on intersection types [54] for the π-calculus and ${}^{e}\pi$. Other works which could be directly relevant involve the use of types in process algebras to study security properties of Web services [37], reasoning about mobile resources [36] and verifying authorization properties [28].

An interesting extension is to assign precise types to trees, describing their schematic structure, rather than regarding them as all belonging to the same sort. In this way, it would be possible to guarantee that a certain transformation conforms to a schema, and to enforce fine-grained access control. A starting point to type the trees could be the work on types and logics for semi-structured data [45, 7, 19]. A further step, which we consider necessary in order to give schema types to X$d\pi$ trees, is to extend the type system with process types [87, 85], which record information about the behaviour of processes. In the case of dynamic Web data, we believe that this extension is needed because the structure of a tree depends also on the result of executing the scripts that it contains, hence a type for a tree must depend on the behavioural types of the scripts it contains.

Given the use of mobile code in our systems, in the absence of trust, we face the problem of protecting a host from a potentially malicious agent. This problem could be tackled by type-checking each agent dynamically entering a location [40] (possibly relying on the ability of a location to infer the type of the agent), or by using the Proof Carrying Code [60] approach (to send the a migrating process along with its type), or by a combination of both techniques.

A type system usually restricts the number of terms that are admissible in a calculus. Hence, the behavioural equivalences can become easier to verify (since there can be less-counter-examples), and some laws that do not hold in the untyped calculus become valid for the typed fragment. An obvious theoretical question arising from the definition of a type system for X$d\pi$, is to understand how the behavioural equivalences are affected by typing, for example along the lines of [64, 39].

Implementation. Several final year students at Imperial College London have based their graduation projects on prototype implementations of X$d\pi$, with encouraging results about the feasibility of our approach in practice. Some have developed it as a stand-alone application, others as a set of library functions to integrate with their favourite programming language. For example, Alex Ahern has developed

a distributed prototype implementation of X$d\pi$ based on XML standards [5]. The implementation embeds processes in XML documents and uses XPath as a query language. Communication between peers is provided through SOAP-based web services and the working space of each location is endowed with a process scheduler based on ideas from PICT [65]. Another strategy, followed by Anthony Cheah [22], consisted in extending the open source implementation of AXML [78] with ideas from X$d\pi$, in particular with support for more flexible service calls and for mobile queries.

In view of a more substantial implementation of X$d\pi$ as a platform for empowering dynamic Web data, we believe that X$d\pi$ could provide the lower-level language spoken directly by peers, while it would be beneficial to define a higher-level language at the application layer rich in derived operations (in the style of those of Section 3.1), with a rigourous semantics given in terms of a translation to X$d\pi$.

Concluding, we think that the presence of a theoretical framework, which can be sometimes seen as limiting the freedom of the implementor to adapt the language specification in view of engineering considerations, has the advantage of providing a solid basis for developing semantically correct programs, and helping the development of program analysis tools.

Bibliography

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *ACM SIGPLAN Notices*, 36(3):104–115, 2001.

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[3] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann, 2000.

[4] Lucia Acciai and Michele Boreale. XPi: A typed process calculus for XML messaging. In *Proceedings of FMOODS'05*, 2005.

[5] Alexander Ahern. A language for updating data. Project report 1441. Imperial College London, Department of Computing Technical Library, 2003.

[6] Omar Benjelloun. Active XML: A data-centric perspective on Web services. Ph.D. Thesis, Universtity of Paris XI, 2002.

[7] Véronique Benzanken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of ICFP '03*, pages 51–63. ACM Press, 2003.

[8] Martin Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College London, 2002.

[9] Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended pi-calculus. In *EXPRESS '00*, volume 39.1 of *ENTCS*. Elsevier Science Publishers, 2000.

[10] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Selected papers of the Second Workshop on Concurrency and compositionality*, pages 217–248. Elsevier Science Publishers, 1992.

[11] Gavin Bierman and Peter Sewell. Iota: a concurrent XML scripting language with application to Home Area Networks. University of Cambridge Technical Report 557, January 2003.

[12] Angela Bonifati and Stefano Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–91, 2000.

[13] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[14] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Alexander Kreutz, Stefan Seltzsam, and Konrad Stocker. ObjectGlobe: Ubiquitous query processing on the internet. To appear in the VLDB Journal: Special Issue on E-Services, 2002.

[15] Allen Brown, Cosimo Laneve, and Greg Meredith. PiDuce: a process calculus with native XML datatypes. Unpublished manuscript, October 2004.

[16] Roberto Bruni, Cosimo Laneve, and Ugo Montanari. Orchestrating transactions in join calculus. In *Proceedings of CONCUR '02*, pages 321–337. Springer Verlag, 2002.

[17] Marco Carbone and Sergio Maffeis. On the expressive power of polyadic synchronisation in π-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.

[18] Luca Cardelli and Rowan Davies. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.

[19] Luca Cardelli and Giorgio Ghelli. A query language based on the ambient logic. In *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 1–22. Springer, 2001.

[20] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[21] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the pi-calculus. In *Proceedings of LICS'05*, pages 92–101. IEEE Computer Society Press, 2005.

[22] Anthony Cheah. Extended Active XML. Project report 1662. Imperial College London, Department of Computing Technical Library, 2005.

[23] Microsoft Corporation. Microsoft biztalk. `http://www.microsoft.com/biztalk/`.

[24] Andrea Ferrara. Web services: a process algebra approach. In *Proceedings of ICSOC '04*, pages 242–251. ACM Press, 2004.

[25] William Ferreira, Matthew Hennessy, and Alan Jeffrey. A theory of weak bisimulation for core cml. In *Proceedings of ICFP '96*, pages 201–212. ACM Press, 1996.

[26] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM Press, 1996.

[27] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proceedings of ICALP '98*, LNCS, pages 844–855. Springer Verlag, 1998.

[28] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In *Proceedings of ESOP'05*, volume 3444 of *LNCS*, pages 141–156. Springer Verlag, April 2005.

[29] Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failures. In *Proceedings of CONCUR 2005*, volume 3653 of *LNCS*, pages 368–382. Springer Verlag, 2005.

[30] Murdoch J. Gabbay. The π-calculus in FM. In *Thirty-five years of Automath*, volume 28, pages 71–123. Kluwer Academic Press, 2003.

[31] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002. Special issue in honour of Rod Burstall.

[32] Philippa Gardner and Sergio Maffeis. Modelling dynamic Web data. In Georg Lausen and Dan Suciu, editors, *Proc. of DBPL'03* (a revised version was presented at *Appsem'04*, Tallinn, 2004), volume 2921 of *LNCS*, pages 130–146. Springer Verlag, 2003.

[33] Philippa Gardner and Sergio Maffeis. Modelling dynamic Web data. Imperial College London Technical Report, Oct 2003.

[34] Philippa Gardner and Sergio Maffeis. Modelling dynamic Web data. *Theoretical Computer Science*, 342:104–131, 2005.

[35] Philippa Gardner, Nobuko Yoshida, Sergio Maffeis, and Alex Ahern. Dynamic Web data: Theory and experiment. Poster at EPSRC e-Science Meeting, Edinburgh, March 2004.

[36] Jens Godskesen, Thomas Hildebrandt, and Vladimiro Sassone. A calculus of mobile resources. In *Proceedings of CONCUR'02*, LNCS, 2002.

[37] Andrew D. Gordon and Riccardo Pucella. Validating a web service security abstraction by typing. In *Proceedings of the 2002 ACM Workshop on XML Security*, pages 18–29, 2002.

[38] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions. In *Proc. of ESOP'03*, volume 2618 of *LNCS*, pages 238–252. Springer Verlag, 2003.

[39] Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. Computer Science Report 01/2002, University of Sussex, 2002.

[40] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. safeDpi: A language for controlling mobile code. In *Proceedings of FoSSaCS 2004*, volume 2987 of *LNCS*, pages 241–256. Springer Verlag, 2004.

[41] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. safedpi: a language for controlling mobile code. *Acta Informatica*, 42(4-5):227–290, 2005.

[42] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *HLCL '98*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier Science Publishers, 1998.

[43] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.

[44] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995.

[45] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[46] Alan Jeffrey and Julian Rathke. Contextual equivalence for higher-order pi-calculus revisited. Computer Science Report 04/2002, University of Sussex, 2002.

[47] Trevor Jim and Dan Suciu. Dynamically distributed query evaluation. In *Proceedings of PODS '01*, pages 28–39. ACM Press, 2001.

[48] Markus Keidl, Stefan Seltzsam, Konrad Stocker, and Alfons Kemper. Serviceglobe: Distributing e-services across the internet. In *VLDB*, pages 1047–1050, 2002.

[49] Alfons Kemper and Christian Wiesner. Hyperqueries: Dynamic distributed query processing on the internet. In *Proceedings of VLDB'01*, pages 551–560, 2001.

[50] Thomas Kistler and Hannes Marais. Webl - a programming language for the web. In *Proceedings of WWW7*, pages 259–270. Elsevier Science Publishers, 1998.

[51] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys.*, 32(4):422–469, 2000.

[52] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *Proceedings of FoSSaCS'05*, number 3441 in LNCS, pages 282–298. Springer Verlag, 2005.

[53] Sergio Maffeis. Process calculi and Web data integration. In *Algebraic Process Calculi: The First Twenty Five Years and Beyond*, BRICS Notes Series, to appear, 2005.

[54] Sergio Maffeis. Sequence types for the pi-calculus. In *ITRS'04*, volume 136 of *ENTCS*, pages 117–132. Elsevier Science Publishers, 2005.

[55] Sergio Maffeis and Philippa Gardner. Behavioural equivalences for dynamic Web data. In *Proceedings of Ifip WCC-TCS'04*, Kluwer Academic Press, pages 535–548, August 2004.

[56] Robin Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.

[57] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[58] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[59] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Exchanging intensional XML data. In *Proceedings of SIGMOD '03*, pages 289–300. ACM Press, 2003.

[60] George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91. Springer Verlag, 1998.

[61] David Neven, Thomas Schwentick, and Dan Suciu. Foundations of semi-structured data. Dagstuhl Seminar Proceedings 05061. Available online from `http://drops.dagstuhl.de/portals/05061/`, 2005.

[62] Vassilis Papadimos and David Maier. Mutant query plans. *Information & Software Technology*, 44(4):197–206, 2002.

[63] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, October 1996.

[64] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Proceedings of POPL '97*, pages 242–255. ACM Press, 1997.

[65] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[66] Arnaud Sahuguet. *ubQL: A Distributed Query Language to Program Distributed Query Systems*. PhD thesis, University of Pennsylvania, 2002.

[67] Arnaud Sahuguet, Benjamin Pierce, and Val Tannen. Distributed Query Optimization: Can Mobile Agents Help? Unpublished draft, 2000.

[68] Arnaud Sahuguet and Val Tannen. ubql, a language for programming distributed query systems. In *Proceedings of webDB'01*, pages 37–42, 2001.

[69] Davide Sangiorgi. Expressing mobility in process algebras: First-order and higher-order paradigms. PhD thesis, University of Edinburgh, 1992.

[70] Davide Sangiorgi. A theory of bisimulation for the pi-calculus. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 127–142, London, UK, 1993. Springer-Verlag.

[71] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[72] Serge Abiteboul, et al. Active XML primer. INRIA Futurs, GEMO Report number 275, 2003.

[73] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of Haskell '02*, pages 1–16. ACM Press, 2002.

[74] Walid Taha. Metaocaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.cs.rice.edu/taha/MetaOCaml/`, 2001.

[75] Bent Thomsen. A calculus of higher order communicating systems. In *Proceedings of POPL '89*, pages 143–154. ACM Press, 1989.

[76] UDDI. Universal Description, Discovery, and Integration of Business for the Web (UDDI) 3.0. `http://www.uddi.org`, 2005.

[77] Asis Unyapoth and Peter Sewell. Nomadic pict: correct communication infrastructure for mobile computation. In *POPL '01*, pages 116–127. ACM Press, 2001.

[78] Gabriel Vasile, Bogdan Cautis, Serge Abiteboul, and Omar Benjelloun. Active xml implementation. `http://forge.objectweb.org/projects/activexml/`, 2004.

[79] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *ICCL Workshop: Internet Programming Languages*, pages 47–77, 1998.

[80] W3C. XML Path Language (XPath) Version 1.0. `http://w3.org/TR/xpath`, 1999.

[81] W3C. Extensible Markup Language (XML) 1.0 (2nd edition). `http://www.w3.org/TR/REC-xml.html`, 2000.

[82] W3C. Web Services Description Language (WSDL) 1.1. `http://w3.org/TR/wsdl`, 2001.

[83] W3C. Web Services Activity. `http://www.w3.org/2002/ws`, 2002.

[84] W3C. Simple Object Acess Protocol (SOAP) Version 1.2. `http://w3.org/TR/SOAP`, 2003.

[85] Nobuko Yoshida. Channel dependent types for higher-order mobile processes. In *Proceedings of POPL '04*, pages 147–160. ACM Press, 2004.

[86] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the π-calculus. *Information and Computation*, 191(2):145–202, 2004.

[87] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. *Information and Computation*, 174(2):143–179, 2002.

Appendix A

Tables

Figure A.1: Function dom for X$d\pi$ networks

$$\mathit{dom}(l\,[\![\,T \parallel P\,]\!]) = \{l\} \qquad \mathit{dom}(\mathbf{0}) = \emptyset$$
$$\mathit{dom}(N \mid M) = \mathit{dom}(N) \cup \mathit{dom}(M) \qquad \mathit{dom}((\nu\, c)N) = \mathit{dom}(N)$$
$$\mathit{dom}(N \mid C_{\mathcal{N}}[-]) = \mathit{dom}(N) \cup \mathit{dom}(C_{\mathcal{N}}[-]) \qquad \mathit{dom}(C_{\mathcal{N}}[-] \mid N) = \mathit{dom}(N) \cup \mathit{dom}(C_{\mathcal{N}}[-])$$
$$\mathit{dom}((\nu\, c)C_{\mathcal{N}}[-]) = \mathit{dom}(C_{\mathcal{N}}[-]) \qquad \mathit{dom}(-) = \emptyset$$

Figure A.2: Function dom for Core X$d\pi$

$$\mathit{dom}(D) = \mathit{domain}(D) \qquad \mathit{dom}(\mathbf{0}) = \emptyset$$
$$\mathit{dom}(\boldsymbol{P} \mid \boldsymbol{Q}) = \mathit{dom}(\boldsymbol{P}) \cup \mathit{dom}(\boldsymbol{Q}) \qquad \mathit{dom}((\nu\, c)\boldsymbol{P}) = \mathit{dom}(\boldsymbol{P})$$
$$\mathit{dom}(\overline{\boldsymbol{l}\cdot\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle) = \{\boldsymbol{l}\} \qquad \mathit{dom}(\boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) = \{\boldsymbol{l}\}$$
$$\mathit{dom}(!\boldsymbol{l}\cdot\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) = \{\boldsymbol{l}\} \qquad \mathit{dom}(\boldsymbol{l}\cdot\mathsf{go}\ \boldsymbol{m}.\boldsymbol{m}\cdot\boldsymbol{P}) = \{\boldsymbol{l}\}$$
$$\mathit{dom}(\boldsymbol{A} \circ \langle\boldsymbol{l}, \widetilde{\boldsymbol{v}}\rangle) = \{\boldsymbol{l}\} \qquad \mathit{dom}(\boldsymbol{l}\cdot\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle) = \{\boldsymbol{l}\}$$
$$\mathit{dom}(-) = \emptyset \qquad \mathit{dom}(C_{\mathcal{S}}[-] \uplus D) = \mathit{dom}(D)$$
$$\mathit{dom}(P \mid C_{\mathcal{P}}[-]) = \mathit{dom}(C_{\mathcal{P}}[-] \mid P) = \mathit{dom}(P) \qquad \mathit{dom}((\nu\, c)C_{\mathcal{P}}[-]) = \emptyset$$

Figure A.3: Full structural congruence for X$d\pi$

$(\nu\, c)\mathbf{0} \equiv \mathbf{0}$ (Struct Res PNil)

$c \notin \mathit{fn}(\boldsymbol{P}) \Longrightarrow \boldsymbol{P} \mid (\nu\, c)\boldsymbol{Q} \equiv (\nu\, c)(\boldsymbol{P} \mid \boldsymbol{Q})$ (Struct Res PPar)

$(\nu\, c)(\nu\, d)\boldsymbol{P} \equiv (\nu\, d)(\nu\, c)\boldsymbol{P}$ (Struct Res PRes)

$\boldsymbol{P} \mid (\boldsymbol{Q} \mid \boldsymbol{Q}') \equiv (\boldsymbol{P} \mid \boldsymbol{Q}) \mid \boldsymbol{Q}'$ (Struct Par PAssoc)

$\boldsymbol{P} \mid \boldsymbol{Q} \equiv \boldsymbol{Q} \mid \boldsymbol{P}$ (Struct Par PComm)

$\boldsymbol{P} \mid \mathbf{0} \equiv \boldsymbol{P}$ (Struct Par PZero)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow (\nu\, c)\boldsymbol{P} \equiv (\nu\, c)\boldsymbol{Q}$ (Struct Cong PRes)

$\boldsymbol{P} \equiv \boldsymbol{P}' \Longrightarrow \boldsymbol{P} \mid \boldsymbol{Q} \equiv \boldsymbol{P}' \mid \boldsymbol{Q}$ (Struct Cong PPar)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow \boldsymbol{a}(\widetilde{\pi}).\boldsymbol{P} \equiv \boldsymbol{a}(\widetilde{\pi}).\boldsymbol{Q}$ (Struct Cong PIn)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow !\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{P} \equiv !\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{Q}$ (Struct Cong P!In)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow \mathsf{go}\,\boldsymbol{m}.\boldsymbol{P} \equiv \mathsf{go}\,\boldsymbol{m}.\boldsymbol{Q}$ (Struct Cong PGo)

$(\nu\, c)\mathbf{0} \equiv \mathbf{0}$ (Struct Res NNil)

$c \notin \mathit{fn}(N) \Longrightarrow N \mid (\nu\, c)M \equiv (\nu\, c)(N \mid M)$ (Struct Res NPar)

$(\nu\, c)(\nu\, d)N \equiv (\nu\, d)(\nu\, c)N$ (Struct Res NRes)

$l\left[\, T \parallel (\nu\, c)P \,\right] \equiv (\nu\, c)l\left[\, T \parallel P \,\right]$ (Struct Res NLoc)

$N \mid (M \mid M') \equiv (N \mid M) \mid M'$ (Struct Par NAssoc)

$N \mid M \equiv M \mid N$ (Struct Par NComm)

$N \mid \mathbf{0} \equiv N$ (Struct Par NZero)

$N \equiv M \Longrightarrow (\nu\, c)N \equiv (\nu\, c)M$ (Struct Cong NRes)

$N \equiv N' \Longrightarrow N \mid M \equiv N' \mid M$ (Struct Cong NPar)

$t \equiv t$ (Struct Refl)

$t \equiv t' \Longrightarrow t' \equiv t$ (Struct Symm)

$t \equiv t''$ *and* $t'' \equiv t' \Longrightarrow t \equiv t'$ (Struct Trans)

$P \equiv Q \Longrightarrow (D, P) \equiv (D, Q)$ (Struct Proc)

Notation: t ranges over $\boldsymbol{P}$ or N.

Figure A.4: Free variables and free names for X$d\pi$

$$
\begin{aligned}
\mathit{fv}(\boldsymbol{E}\triangleright\boldsymbol{T}) &= \mathit{fv}(\boldsymbol{E}) \cup \mathit{fv}(\boldsymbol{T}) & \mathit{fv}(\varnothing) &= \emptyset \\
\mathit{fv}(x) &= \{x\} & \mathit{fv}(\mathsf{a}[\,\boldsymbol{V}\,]) &= \mathit{fv}(\boldsymbol{V}) \\
\mathit{fv}(\boldsymbol{p}@\boldsymbol{l}) &= \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\boldsymbol{p}) & \mathit{fv}(l) &= \emptyset \\
\mathit{fv}((\widetilde{\pi})\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{P}) \setminus \mathit{fv}(\widetilde{\pi}) & \mathit{fv}(\boldsymbol{0}) &= \emptyset \\
\mathit{fv}(\boldsymbol{P} \mid \boldsymbol{Q}) &= \mathit{fv}(\boldsymbol{P}) \cup \mathit{fv}(\boldsymbol{Q}) & \mathit{fv}((\nu\, c)\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{P}) \\
\mathit{fv}(\overline{\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle) &= \mathit{fv}(\boldsymbol{c}) \cup \mathit{fv}(\widetilde{\boldsymbol{v}}) & \mathit{fv}(\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{c}) \cup (\mathit{fv}(\boldsymbol{P}) \setminus \mathit{fv}(\widetilde{\pi})) \\
\mathit{fv}(!\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{c}) \cup (\mathit{fv}(\boldsymbol{P}) \setminus \mathit{fv}(\widetilde{\pi})) & \mathit{fv}(\mathsf{go}\ \boldsymbol{l}.\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\boldsymbol{P}) \\
\mathit{fv}(\boldsymbol{A} \circ \langle\widetilde{\boldsymbol{v}}\rangle) &= \mathit{fv}(\boldsymbol{A}) \cup \mathit{fv}(\widetilde{\boldsymbol{v}}) & \mathit{fv}(\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle) &= \mathit{fv}(\boldsymbol{p}) \cup \mathit{fv}(\boldsymbol{c}) \\
\mathit{fv}(c) = \mathit{fv}(\mathsf{c}) &= \emptyset
\end{aligned}
$$

$$
\left.\begin{aligned} \mathit{fn}(T) = \mathit{fn}(E) = \mathit{fn}(A) \\ \mathit{fn}(p) = \mathit{fn}(l) = \mathit{fn}(x) \end{aligned}\right\} = \emptyset \qquad \mathit{fn}(\boldsymbol{0}) = \emptyset
$$

$$
\begin{aligned}
\mathit{fn}(\boldsymbol{P} \mid \boldsymbol{Q}) &= \mathit{fn}(\boldsymbol{P}) \cup \mathit{fn}(\boldsymbol{Q}) & \mathit{fn}((\nu\, c)\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{P}) \setminus \{c\} \\
\mathit{fn}(\overline{\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle) &= \mathit{fn}(\boldsymbol{c}) \cup \mathit{fn}(\widetilde{\boldsymbol{v}}) & \mathit{fn}(\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{c}) \cup \mathit{fn}(\boldsymbol{P}) \\
\mathit{fn}(!\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{c}) \cup \mathit{fn}(\boldsymbol{P}) & \mathit{fn}(\mathsf{go}\ \boldsymbol{l}.\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{P}) \\
\mathit{fn}(\boldsymbol{A} \circ \langle\widetilde{\boldsymbol{v}}\rangle) &= \mathit{fn}(\widetilde{\boldsymbol{v}}) & \mathit{fn}(\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle) &= \mathit{fn}(\boldsymbol{c}) \\
\mathit{fn}(c) &= \{c\} & \mathit{fn}(\mathsf{c}) &= \emptyset \\
\mathit{fn}(l\,[\,T \parallel P\,]) &= \mathit{fn}(P) & \mathit{fn}(\boldsymbol{0}) &= \emptyset \\
\mathit{fn}(N \mid M) &= \mathit{fn}(N) \mid \mathit{fn}(M) & \mathit{fn}((\nu\, c)N) &= \mathit{fn}(N) \setminus \{c\}
\end{aligned}
$$

Figure A.5: Full structural congruence for Core X$d\pi$

$(\nu\, c)\mathbf{0} \equiv \mathbf{0}$ (CStruct Res PNil)

$c \notin \mathit{fn}(\boldsymbol{P}) \Longrightarrow \boldsymbol{P} \mid (\nu\, c)\boldsymbol{Q} \equiv (\nu\, c)(\boldsymbol{P} \mid \boldsymbol{Q})$ (CStruct Res PPar)

$(\nu\, c)(\nu\, d)\boldsymbol{P} \equiv (\nu\, d)(\nu\, c)\boldsymbol{P}$ (CStruct Res PRes)

$\boldsymbol{P} \mid (\boldsymbol{Q} \mid \boldsymbol{Q}') \equiv (\boldsymbol{P} \mid \boldsymbol{Q}) \mid \boldsymbol{Q}'$ (CStruct Par Assoc)

$\boldsymbol{P} \mid \boldsymbol{Q} \equiv \boldsymbol{Q} \mid \boldsymbol{P}$ (CStruct Par Comm)

$\boldsymbol{P} \mid \mathbf{0} \equiv \boldsymbol{P}$ (CStruct Par Zero)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow (\nu\, c)\boldsymbol{P} \equiv (\nu\, c)\boldsymbol{Q}$ (CStruct Cong Res)

$\boldsymbol{P} \equiv \boldsymbol{P}' \Longrightarrow \boldsymbol{P} \mid \boldsymbol{Q} \equiv \boldsymbol{P}' \mid \boldsymbol{Q}$ (CStruct Cong Par)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow \boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{P} \equiv \boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{Q}$ (CStruct Cong In)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow !\boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{P} \equiv !\boldsymbol{l}\cdot\boldsymbol{a}(\widetilde{\pi}).\boldsymbol{Q}$ (CStruct Cong !In)

$\boldsymbol{P} \equiv \boldsymbol{Q} \Longrightarrow \boldsymbol{l}\cdot\mathsf{go}\,\boldsymbol{m}.\boldsymbol{P} \equiv \boldsymbol{l}\cdot\mathsf{ping}\,\boldsymbol{m}.\boldsymbol{Q}$ (CStruct Cong Go)

$P \equiv Q \Longrightarrow (D, P) \equiv (D, Q)$ (CStruct Proc)

$(\nu\, c)\mathbf{0} \equiv \mathbf{0}$ (Struct Res CNil)

$c \notin \mathit{fn}(K) \Longrightarrow K \mid (\nu\, c)K' \equiv (\nu\, c)(K \mid K')$ (Struct Res CPar)

$(\nu\, c)(\nu\, d)K \equiv (\nu\, d)(\nu\, c)K$ (Struct Res CRes)

$K \mid (K' \mid L) \equiv (K \mid K') \mid L$ (Struct Par CAssoc)

$K \equiv K' \Longrightarrow K' \equiv K$ (Struct Par CComm)

$K \mid \mathbf{0} \equiv K$ (Struct Par CZero)

$K \equiv K' \Longrightarrow (\nu\, c)K \equiv (\nu\, c)K'$ (Struct Cong CRes)

$K \equiv L \Longrightarrow K \mid K' \equiv L \mid K'$ (Struct Cong CPar)

$t \equiv t$ (CStruct Refl)

$t \equiv t' \Longrightarrow t' \equiv t$ (CStruct Symm)

$t \equiv t''$ *and* $t'' \equiv t' \Longrightarrow t \equiv t'$ (CStruct Trans)

Notation: t ranges over $\boldsymbol{P}$ or K.

Figure A.6: Free variables and free names for Core X$d\pi$

$$
\begin{aligned}
\mathit{fv}((x,\widetilde{\pi})\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{P}) \setminus \mathit{fv}(x,\widetilde{\pi}) & \qquad \mathit{fv}(\boldsymbol{0}) &= \emptyset \\
\mathit{fv}(\boldsymbol{P} \mid \boldsymbol{Q}) &= \mathit{fv}(\boldsymbol{P}) \cup \mathit{fv}(\boldsymbol{Q}) & \qquad \mathit{fv}((\nu\, c)\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{P}) \\
\mathit{fv}(\boldsymbol{l}{\cdot}\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{l}{\cdot}\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\boldsymbol{c}) \cup (\mathit{fv}(\boldsymbol{P}) \setminus \mathit{fv}(\widetilde{\pi})) & \qquad \mathit{fv}(\overline{\boldsymbol{l}{\cdot}\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle) &= \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\boldsymbol{c}) \cup \mathit{fv}(\widetilde{\boldsymbol{v}}) \\
\mathit{fv}(!\boldsymbol{l}{\cdot}\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{l}{\cdot}\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\boldsymbol{c}) \cup (\mathit{fv}(\boldsymbol{P}) \setminus \mathit{fv}(\widetilde{\pi})) & \qquad \mathit{fv}(\boldsymbol{l}{\cdot}\mathsf{go}\ \boldsymbol{m}.\boldsymbol{m}{\cdot}\boldsymbol{P}) &= \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\boldsymbol{m}) \cup \mathit{fv}(\boldsymbol{P}) \\
\mathit{fv}(\boldsymbol{A} \circ \langle \boldsymbol{l}, \widetilde{\boldsymbol{v}}\rangle) &= \mathit{fv}(\boldsymbol{A}) \cup \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\widetilde{\boldsymbol{v}}) & \qquad \mathit{fv}(\boldsymbol{l}{\cdot}\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle) &= \mathit{fv}(\boldsymbol{l}) \cup \mathit{fv}(\boldsymbol{p}) \cup \mathit{fv}(\boldsymbol{c})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fn}(\boldsymbol{0}) &= \emptyset & \qquad \mathit{fn}(\boldsymbol{P} \mid \boldsymbol{Q}) &= \mathit{fn}(\boldsymbol{P}) \cup \mathit{fn}(\boldsymbol{Q}) \\
\mathit{fn}((\nu\, c)\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{P}) \setminus \{c\} & \qquad \mathit{fn}(\overline{\boldsymbol{l}{\cdot}\boldsymbol{c}}\langle\widetilde{\boldsymbol{v}}\rangle) &= \mathit{fn}(\boldsymbol{c}) \cup \mathit{fn}(\widetilde{\boldsymbol{v}}) \\
\mathit{fn}(\boldsymbol{l}{\cdot}\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{c}) \cup \mathit{fn}(\boldsymbol{P}) & \qquad \mathit{fn}(!\boldsymbol{l}{\cdot}\boldsymbol{c}(\widetilde{\pi}).\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{c}) \cup \mathit{fn}(\boldsymbol{P}) \\
\mathit{fn}(\boldsymbol{l}{\cdot}\mathsf{go}\ \boldsymbol{m}.\boldsymbol{m}{\cdot}\boldsymbol{P}) &= \mathit{fn}(\boldsymbol{P}) & \qquad \mathit{fn}(\boldsymbol{A} \circ \langle \boldsymbol{l}, \widetilde{\boldsymbol{v}}\rangle) &= \mathit{fn}(\widetilde{\boldsymbol{v}}) \\
\mathit{fn}(\boldsymbol{l}{\cdot}\mathsf{req}_{\boldsymbol{p}}\langle\boldsymbol{c}\rangle) &= \mathit{fn}(\boldsymbol{c}) & \qquad \mathit{fn}((D, P)) &= \mathit{fn}(P)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fv}(\boldsymbol{K} \mid \boldsymbol{K}') &= \mathit{fv}(\boldsymbol{K}) \cup \mathit{fv}(\boldsymbol{K}') & \qquad \mathit{fv}((\nu\, c)\boldsymbol{K}) &= \mathit{fv}(\boldsymbol{K}) \\
\mathit{fv}(\langle k \Leftarrow A\rangle) &= \emptyset & \qquad \mathit{fv}(k) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fn}(\boldsymbol{K} \mid \boldsymbol{K}') &= \mathit{fn}(\boldsymbol{K}) \cup \mathit{fn}(\boldsymbol{K}') & \qquad \mathit{fn}((\nu\, c)\boldsymbol{K}) &= \mathit{fn}(\boldsymbol{K}) \\
\mathit{fn}(\langle k \Leftarrow A\rangle) &= \{k\} & \qquad \mathit{fn}(k) &= \{k\}
\end{aligned}
$$