MIKADO
Mobile Calculi based on Domains

MIKADO Global Computing Project
IST-2001-32222

System Behaviour and Reasoning in the Presence of Failure

MIKADO Deliverable D2.3.2

Editor : A. FRANCALANZA (U. OF SUSSEX)

Authors : A. FRANCALANZA (U. OF SUSSEX)

Contributors : A. FRANCALANZA, M. HENNESSY (U. OF SUSSEX)
R. DE NICOLA, D. GORLA (U. DI FIRENZE)
R. PUGLIESE (U. DI ROMA LA SAPIENZA)

Classification : Public
Deliverable no. : D2.3.2
Reference : RR/WP2/3
Date : December 2004

Project funded by the European Community under the "Information Society Technologies" Programme (1998–2002)

D2.3.3: System Behaviour and Reasoning in the Presence of Failure

Adrian Francalanza
adrianf@sussex.ac.uk

January 11, 2005

1 Introduction

It is generally accepted that location transparency is hard to attain over Wide Area Networks [4] : one of the main reasons why this is hard to attain is *partial failure*, which reveals the underlying stucture of the distribution of computation. Over the last decade, various distributed process calculi have arisen to capture and study behaviour of distributed programs in the presence of failure [2, 1, 16, 3, 17] . In this report, we overview the work done in the Mikado project [15]along this line of research. To facilitate our comparison between work carried out by different authors, we first give a brief classification of the different aspects of failure in § 2, followed by the work on distributed calculi in § 3.

2 Systems, Faults, Partial Failure and Fault Detectors

Distributed programs in process calculi are sometimes referred to as *systems*, consisting of *processes* distibuted over a network. This network can be represented in various ways, usually involving notions of *locations* subject to a *location structure*; this structure can denote dependencies among the locations or restrictions for communication or migration across locations. The simplest structure is *flat*, where every location is interconnected with one another; a *hierarchical* structure is another structure and often denotes location dependencies such as a location containing other locations; a *graph* location structure is perhaps the most general and can be used to describe networks where not every location is interconnected. In this setting, the behaviour of a system is dependent on the network representation over which it is running; for instance, if a network is represented by a graph location structure, two processes located at distinct locations would *only* be able to communicate directly with one another, if there is a (direct) link between the two hosting locations.

A network representation may carry a notion of *state*, such as the liveness of nodes. *Faults*, that is defects in the network, are represented as changes the network state; for instance a location's state may change from alive to dead. Faults may affect the behaviour of a system, and this change in behaviour is called *failure* [11]. In a distributed

setting, the effect of a fault is often limited to a subset of the system, which is referred to as *partial failure*. For example, if a location dies, all the processes at that location stop executing, a type of failure called fail-stop [18]. Other classes of failure may be observed in systems, depending on the underlying fault. Failures may be *permanent* or *transient* depending on whether a the faulty component recovers its state or not. A fault may also affect part of a location, which may lead to *byzantine* failures at that location [14].

Apart from failures, faults can also be *actively observed* in systems through *fault detectors*; these detectors are used to trigger *fault recovery* so as to minimise failure. The most popular classification of fault detectors [5] is based on the notion of *correctness* and *completness*. The simplest and most powerful is the *perfect fault detector*; this however turns out to be hard to implement in a Wide Area Network setting, which is inherently asynchronous and does not allow tight synchronisations between locations [8]. Nevertheless, perfect fault detection is still considered useful from a theoretical point of view.

3 Work on Distributed Process Calculi with failures

In the Mikado project, there have been two efforts that have addressed the problem of prcess calculi behaviour in the presence of failure. We here outline both with respect to the concepts described in § 2.

3.1 tKlaim: Topological Klaim

tKlaim [7], or *topological Klaim*, is an extension of *linda-like* distributed language called cKlaim [6] where a graph based network representation is introduced and direct interaction across two locations is allowed only if there exists a link between the two locations.

3.1.1 The Syntax

The top level syntax of the language, that is *tKlaim nets*, may take the form:

$$l :: P \parallel (\nu k)(\, l :: \mathbf{in}(T)@k.Q \parallel k : \langle t \rangle \parallel \{l \leftrightarrow k\}\,) \tag{1}$$

where location l is a *free* location hosting two processes, P and $\mathbf{in}(T)@k.Q$, k is a *scoped* location, hosting one data element, $\langle t \rangle$ and a scope that extends also to the process $\mathbf{in}(T)@k.Q$ located at l. The remaining sub-term in (1), $\{l \leftrightarrow k\}$, represents the fact that l and k are bi-directionally connected; this allows $\mathbf{in}(T)@k.Q$ to input, directly from l, the data $\langle t \rangle$, located at k.

3.1.2 Failures modelled

A form of *fail-stop process/data failure* is described in tKlaim through the reduction rule:

$$\text{(R-FailN)} \qquad l :: C \longrightarrow l :: \mathbf{0}$$

This failure does not translate to location failure, where typically, *every* process/data at a particular dead location is affected. tKlaim does not explicitly represent the state of its locations, and as a result, (R-FailN) cannot be linked to the underlying state of a faulty location. Thus, the tKlaim net in (1) can reduce to:

$$l :: P \parallel (\nu k)(\, l :: \mathbf{in}(T)@k.Q \parallel \ldots) \;\longrightarrow\; l :: \mathbf{0} \parallel (\nu k)(\, l :: \mathbf{in}(T)@k.Q \parallel \ldots)$$

where process P at l fails while the other process at l, $\mathbf{in}(T)@k.Q$, remains unaffected.

On the other hand, the calculus manages to describe well *fail-stop link failures* through the reduction rule :

$$\text{(R-FailC)} \qquad \{l \leftrightarrow k\} \;\longrightarrow\; \mathbf{0}$$

which induces a link fault in the network representation. Thus, the tKlaim net in (1) can reduce to:

$$\begin{aligned} l :: P \parallel (\nu k)(\, l :: \mathbf{in}(T)@k.Q \parallel k : \langle t \rangle \parallel \{l \leftrightarrow k\}) \;\longrightarrow\; \\ l :: P \parallel (\nu k)(\, l :: \mathbf{in}(T)@k.Q \parallel k : \langle t \rangle \parallel \mathbf{0}) \end{aligned}$$

and as a consequence, a *link failure* is observed, where the process $l :: \mathbf{in}(T)@k.Q$ cannot input the data $\langle t \rangle$ at k, since the bidirectional link does not exist anymore.

tKlaim does not provide any fault detection construct, based on the rationale that in practice, they are hard to attain in asynchronous networks.

3.1.3 Results Obtained

In [7], the authors present a labelled transition system together with a bisimulation for tKlaim and prove its soundness with respect to may testing equivalence. The authors also provide various examples implemented in tKlaim justifying their design choices such as routing algorithms and the k-agreement algorithm.

3.2 Dπ_f: Dπ with Location and Link failure

Dπ_f[10, 9], is an extension of Dπ [12, 13] with a network representation and a number of constructs for inducing and observing faults.

3.2.1 The Syntax

The two tiered syntax of Dπ_f has the form:

$$\Pi \triangleright N$$

where the behaviour of a Dπ system N is subject to a graph based network representation Π, denoting the state of existing locations and bi-directional links between these locations; stated otherwise, the calculus represents both *node* and *link faults* and the respective failure. For instance, the Dπ_f term:

$$\Pi \triangleright l[\![\mathsf{go}\ k.P]\!] \mid l[\![\mathsf{go}\ m.\mathsf{go}\ k.Q]\!] \tag{2}$$

describes two processes, $\mathsf{go}\ k.P$ and $\mathsf{go}\ m.\mathsf{go}\ k.Q$, both located at l, and both trying to reach location k along different routes. This system behaves differently, depending on the state of the network Π: if locations l, k, m and links $l \leftrightarrow k, l \leftrightarrow m, m \leftrightarrow k$ are all alive, then both P and Q will eventually reach k; if the link $l \leftrightarrow k$ is dead, this yeilds a *link failure*, whereat the system level, location k is unreachable from certain locations and only Q can reach k; if location k is dead, this yeilds to a *location failure*, where at the system level, any process at k is halted and k is unreachable from any other location. Another salient point of the Dπ_f syntax is that state information regarding a node is encode as type information when the location is scoped. For instance, the Dπ_f term:

$$\Pi \triangleright M \mid (\nu k : \mathbf{loc}[\mathbf{a}, \{l, m\}])(N) \tag{3}$$

denotes a scoped location k whose scope extends throughout the subsystem N(but not M). The type associated with it, $\mathbf{loc}[\mathbf{a}, \{l, m\}]$, states that it is alive, denoted by the tag **a**, and connected to the free locations l and m, denoted by the set $\{l, m\}$.

3.2.2 Failured modelled

Dπ_f systems are also extended with the constructs kill and $\mathsf{break}\ l$, which *inject* permanent faults at the network level. These constructs enable contexts to change, in controlled fashion, the state of the network at runtime. The reduction rules for these constructs are:

(r-kill)
$$\frac{\Pi \vdash l : \mathbf{live}}{\Pi \triangleright l[\![\mathsf{kill}]\!] \longrightarrow (\Pi - l) \triangleright l[\![\mathbf{0}]\!]}$$

(r-brk)
$$\frac{\Pi \vdash l : \mathbf{live}, l \leftrightarrow k : \mathbf{live}}{\Pi \triangleright l[\![\mathsf{break}\ k]\!] \longrightarrow (\Pi - l \leftrightarrow k) \triangleright l[\![\mathbf{0}]\!]}$$

The final construct extension to Dπ_f is a ping construct that could be used to construct *perfect failure detection*. The reduction rules for this conditional construct are

(r-ping)
$$\frac{\Pi \vdash k : \mathbf{live}, l \leftrightarrow k : \mathbf{live}}{\Pi \triangleright l[\![\mathsf{ping}\ (k)P\lceil Q\rceil]\!] \longrightarrow \Pi \triangleright l[\![P]\!]}$$

(r-nping)
$$\frac{\Pi \nvdash k : \mathbf{live}, l \leftrightarrow k : \mathbf{live}}{\Pi \triangleright l[\![\mathsf{ping}\ (k)P\lceil Q\rceil]\!] \longrightarrow \Pi \triangleright l[\![Q]\!]}$$

Despite the fact that such a construct is hard to attain in practice, the power of the ping construct turn out to be crucial for the completness of the subsequent theory of Dπ_f, since it gives observing contexts the required power to detect changes in the network representation. Another important aspect of ping is that the failure observed may not correspond directly to the fault that causes it; at most, ping can only observe a failure where a node has become completely inaccessible, which can be caused by iether a node fault or multiple faults to all the links connected to that node.

3.2.3 Results

In [10], the authors give a labelled transition system and a bisimulation that is sound and complete with respect to a reduction barbed congruence. A novel aspect of the bismulation is the notion of partial views which handles scope extruded location names that are inccessible to the observer: a scope extruded location is inaccessible when

iether it is dead or there does not exist a path of free locations that lead to that location. In such cases, the observer is prohibited from observing the state of an scope extruded inaccessible location in an labelled transition system term. Fault tolerance in Dπ_f is studied in [9], where a formal definition of faut tolerance is given together with bisimulation-based tractable techniques for verifying fault tolerance.

4 Conclusions

This report overviewed the work done in the Mikado project in the field of process calculi behaviour in the presence of failures. We first categorised the different aspect of failure, from faults to fault tolerance, which facilitated the subsequent presentation of two calculi dealing with failure, tKlaim and Dπ_f.

References

[1] Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In D. Garlan and D. Le Métayer, editors, *Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION'97)*, volume 1282, pages 374–391, Berlin, Germany, 1997. Springer-Verlag.

[2] Roberto M. Amadio and Sanjiva Prasad. Localities and failures. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 14, 1994.

[3] Martin Berger. Basic theory of reduction congruence for two timed asynchronous π-calculi. In *Proc. CONCUR'04*, 2004.

[4] Luca Cardelli. Mobile ambient synchronisation. Technical Report 1997-013, Digital SRC, 1997.

[5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. Technical Report 07/2004, Dip. di Informatica, Univ. di Roma "La Sapienza", 2004.

[7] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. Global computing in a dynamic network of tuple spaces. Draft Copy.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[9] Adrian Francalanza and Matthew Hennessy. Failure and fault tollerance in a distributed pi-calculus. Draft Copy (Nov 2004).

[10] Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failures. Draft Copy (Oct 2004).

[11] Felix C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[12] Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 322:615–669, 2004.

[13] Matthew Hennessy and Julian Rathke. Typed behavioural equivalences for processes in the presence of subtyping. *Mathematical Structures in Computer Science*, 14:651–684, 2004.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[15] Mikado: *GC Project*, IST-2001-32222. `http://mikado.di.fc.ul.pt`.

[16] Nestmann, Fuzzati, and Merro. Modeling consensus in a process calculus. In *CONCUR: 14th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2003.

[17] James Riely and Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 226:693–735, 2001.

[18] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3):222–238, 1983.

Global Computing in a Dynamic Network of Tuple Spaces

occo e N co n e e o ' o o e e
p en o d e e n o c n e d enze
p en o d n o c n e d o L p enz_

Abstract. e p e en c c n p ed y LAIM o e n e e e e p c p oce d on nd node n e connec on e o e ope on p oce o y nd ync ono co n c on o d ed p e p ce e Þ p e en c e n e e connec on e e e nd e en e en c e c e o o o e d nced e e o o co p n e **failures** nd **dynamically evolving connections** n e c e n e e o o o pec y o e non o co p n pp c on o e o e e e po e e n c eo y ed on n o e on e ence o e on y e p ope e o e p opo ed c e d e

1 Introduction

o n co p on n c e e o y o o e n n o e ce eco e one o e n e n Co p e c ence e c en e co e o e nece y o de n e e e e co n c on co ope on o y e o ce e ec y p cy e e c n e n e e de nd nd n ee c n e e y d e en o e ny d e en co ponen A ey e e deÞn on o nno e eo e co p on p d n c ec n nd p e en on ec n e o e de n e z on dep oy en nd n e en o o co p on en on en nd e pp c on Qn e n c de e e e e n e o o co p n o d e e pped p e ppo **network awareness** e oc on c n e e p c y e e enced nd ope on c n e e o e y n o ed **disconnected operations** e code c n e o ed o one oc on o e o e nd e o e y e ec ed **ßexible communication mechanisms** e d ed epo o e o n con en d d e e d nd **remote operations** e ync ono e o e co n c on Qn e o nd on de e de nd on e de e op en o oo nd ec n e o d e nd o y o y e o n yze e e o nd o de on e e con o nce o en pec Þc on C e y c e n c eo e o d eßec e o e ed d nc e e e o o y e n e e de e oped e e n c eo y o ode c ed c LAIM e o n o o o oppo e o ec e Qn one nd e e e p o n n eX LAIM ßed ed p o n n e o o co p e ed on LAIM on e o e nd e e e π c c e en e y eco n zed n co on deno n o o c c o o y c LAIM c n e o o n o e π c c p oce d on nd o y e o e ope on nd ync ono co n c on o d ed epo o e

Moreover, it is equipped with semantic theories that can be used as means to state and prove properties of programs written in X-Klaim.

In this paper, we extend cKlaim with new primitives to model the interconnection structure underlying a net; the resulting formalism is called tKlaim (*topological* Klaim) and its main feature is that only directly connected nodes can directly interact. This choice reflects a concrete feature of global computers: only physically connected machines can exchange information. Sophisticated *routing algorithms* are then needed to enable remote operations between nodes that are not directly connected; however, this aspect is invisible to a user because it is transparently supported by the underlying network architecture.

To softly introduce the reader to our language, we start in Section 2 by presenting a very basic model where inter-node connections are explicitly programmable but fixed at the outset. This scenario is very close to LANs: indeed, physical connections are reliable and immutable (or change very rarely). Section 3 presents a possible use of the language to program communications between not adjacent machines. In particular, we present a routing messenger agent and prove soundness of its behaviour by exploiting *may testing* [12], an intuitive notion of observational equivalence.

We then present two variations of this basic formalism. In Section 4, we enrich the language with different forms of failures, another key feature of global computers. We start with a scenario where only nodes and node components (i.e., data or processes) can fail and use it to establish soundness of a distributed fault-tolerant protocol, the *'k-set agreement'* [8]; then, we briefly present a way to also encompass link failures. The second variation of the basic framework is in Section 5, where links can be dynamically changed by processes. The use of the language with both link failures and dynamic connections is exemplified by programming two routing scenarios and to establish their soundness.

Section 6 concludes the paper with a discussion on related work. More technical material is in Appendix A, where we give a sound (bisimulation-based) proof technique for may testing that can be used for proving properties of the considered examples. Appendixes B and C contain some of the proofs relative to the examples that have been removed from the body of the paper to save space.

2 The Language

2.1 Syntax

The syntax of tKlaim, given in Table 1, is parameterized with respect to the following syntactic sets, which we assume to be countable and pairwise di[redacted] , of *localities*, ranged over by l; , of *locality variables*, ranged over by u; , of *basic values*, ranged over by V; , of *basic variables*, ranged over by x; , of *process variables*, ranged over by X. We use to rang[redacted] .

The exact syntax of *expressions*, e, is deliberately not specified; we just assume that expressions contain, at least, basic values and variables. *Localities*, l, are the addresses (i.e. network references) of nodes. *Tuples*, t, are sequences of expressions, localities or locality variables. *Templates*, T, are used to select tuples: in particular, $!\,x$ and $!\,u$, that we call *formal Þelds*, are used to bind variables to values.

Nets:	Components:
$N ::= 0 \quad l :: C \quad \{l_1 \quad l_2\} \quad (\nu l)N \quad N_1 \; N_2$	$C ::= P \quad t \quad C_1 \mid C_2$
Processes:	**Tuples:**
$P ::= \mathsf{nil} \quad a.P \quad P_1 \mid P_2 \quad X \quad \mathsf{rec}\, X.P$	$t ::= e \quad \ell \quad t_1, t_2$
Actions:	
$a ::= \mathsf{in}(T)@\ell \quad \mathsf{read}(T)@\ell \quad \mathsf{out}(t)@\ell \quad \mathsf{eval}(P)@\ell \quad \mathsf{new}(l)$	
Templates:	**Expressions:**
$T ::= e \quad !x \quad \ell \quad !u \quad T_1, T_2$	$e ::= V \quad x \quad \ldots$

Table 1: KLAIM Syntax

Processes, ranged over by $P, Q, R, \ldots$, are the KLAIM active computational units and may be executed concurrently either at the same locality or at different localities. They are built up from the terminated process nil and from the basic actions by using prefixing, parallel composition and recursion. *Actions* permit removing/accessing/adding tuples from/to tuple spaces, activating new threads of execution and creating new nodes. Action new is not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take effect.

Nets, ranged over by $N, M, \ldots$, are finite collections of nodes and inter-node connections. A *node* is a pair $l :: C$, where locality l is the address of the node and C is the (parallel) component located at l. *Components*, ranged over by $C, D, \ldots$, can be either processes or data, denoted by t. *Connections*, or *links*, are pairs of node addresses $\{l_1 \quad l_2\}$ stating that the nodes with address l_1 and l_2 are directly linked via a physical medium. Connections are bidirectional and can be duplicated, since the same two nodes could be connected by using different physical media; hence the net $N \quad \{l_1 \quad l_2\} \quad \{l_1 \quad l_2\}$ is allowed. In the net $(\nu l)N$, the scope of the name l is private to N; the intended effect is that if one considers the net $N_1 \quad (\nu l)N_2$ then locality l of N_2 cannot be immediately referred to from within N_1.

Names (i.e. localities and variables) occurring in KLAIM processes and nets can be *bound*. More precisely, prefixes $\mathsf{in}(T)@\ell.P$ and $\mathsf{read}(T)@\ell.P$ bind T's formal fields in P; prefix $\mathsf{new}(l).P$ binds l in P, and, similarly, net restriction $(\nu l)N$ binds l in N; finally, $\mathsf{rec}\, X.P$ binds X in P. A name that is not bound is called *free*. The sets $fn(\ )$ and $bn(\ )$ (respectively, of free and bound names of a term) are defined accordingly. The set $n(\ )$ of names of a term is the union of its sets of free and bound names. As usual, we say that two terms are *alpha-equivalent* if one can be obtained from the other by renaming bound names. In the sequel, we shall work with terms whose bound names are all distinct and different from the free ones. Moreover, as usual, we shall only consider *closed* terms, i.e. processes and nets without free variables.

Notation 1. We write $A \triangleq W$ to mean that A is of the form W; this notation is used to assign a symbolic name A to the term W. We shall use notation to denote sets of objects (e.g. l is a set of names). We shall sometimes write $\mathsf{in}()@l$, $\mathsf{out}()@l$ and to mean that the argument of the actions or the datum are irrelevant. Finally, we omit

$match(l; l) = \epsilon$	$match(V; V) = \epsilon$
$match(!u; l) = [{}^{l}/_{u}]$	$match(!x; V) = [{}^{V}/_{x}]$
$\dfrac{match(T_1; t_1) = \sigma_1 \quad match(T_2; t_2) = \sigma_2}{match(T_1, T_2; t_1, t_2) = \sigma_1 \circ \sigma_2}$	

Table 2. Pattern Matching Function

trailing occurrences of process **nil** and write $\Pi_{j\in J}\ W_j$ for the parallel composition (both '|' and '||') of terms (components or nets, resp.) W_j.

2.2 Operational Semantics

tKlaim operational semantics is given in terms of a structural congruence and a reduction relation. The *structural congruence*, $\equiv$, identifies nets which intuitively represent the same net. It is inspired to π-calculus's structural congruence (see, e.g., [19]) and includes laws stating that '||' is commutative, associative and has $\mathbf{0}$ as identity element, laws equating alpha-equivalent nets, laws regulating scope extensions and commutativity of restrictions, and laws allowing to freely fold/unfold recursive processes. Moreover, the following laws are peculiar to our setting:

(Abs)	(Clone)
$l :: C \ \equiv\ l :: C \vert \mathbf{nil}$	$l :: C_1 \vert C_2 \ \equiv\ l :: C_1 \parallel l :: C_2$
(BiDir)	(LinkNode)
$\{l_1 \leftrightarrow l_2\} \ \equiv\ \{l_2 \leftrightarrow l_1\}$	$\{l_1 \leftrightarrow l_2\} \ \equiv\ \{l_1 \leftrightarrow l_2\} \parallel l_1 :: \mathbf{nil}$
(Self)	(RNode)
$l :: \mathbf{nil} \ \equiv\ \{l \leftrightarrow l\}$	$\dfrac{N \equiv (\nu \widetilde{l})(N' \parallel l' :: P) \qquad l \in fn(P)}{(\nu l)N \equiv (\nu l)(N \parallel \{l \leftrightarrow l'\})}$

Law (Abs) states that **nil** is the identity for '|', while law (Clone) turns a parallel between co-located components into a parallel between nodes (thus, it is also used to achieve commutativity and associativity of '|'). Law (BiDir) states that links are bidirectional, law (Self) states that nodes are self-connected and law (LinkNode) states that, if there exists a link $\{l_1 \leftrightarrow l_2\}$, then nodes l_1 and l_2 do exist. Finally, law (RNode) says that any restricted name can be used as the address of a node, always available to the processes knowing that name. Indeed, we consider restricted names as private network addresses, whose corresponding nodes can be activated when needed, and successively deactivated, by the owners of the resource (i.e. the nodes knowing its name).

The reduction relation is given in Table 3 and relies on two auxiliary functions: $\mathcal{E}[\![\ \_\ ]\!]$ and $match(\_; \_)$. The *tuple/template evaluation* function, $\mathcal{E}[\![\ \_\ ]\!]$, evaluates componentwise the expressions occurring within the tuple/template $\_$; its definition is simple and, thus, omitted. The *pattern matching* function, $match(\_; \_)$, verifies the compliance of a tuple w.r.t. a template and associates values to variables bound in the template.

$$\text{(R-Out)}\quad \frac{\mathcal{E}[\![\, t \,]\!] = t'}{l :: \mathbf{out}(t)@l'.P \parallel \{l \leftrightarrow l'\} \;\longmapsto\; l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t' \rangle}$$

$$\text{(R-Eval)}\quad \frac{}{l :: \mathbf{eval}(P_2)@l'.P_1 \parallel \{l \leftrightarrow l'\} \;\longmapsto\; l :: P_1 \parallel \{l \leftrightarrow l'\} \parallel l' :: P_2}$$

$$\text{(R-In)}\quad \frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel \{l \leftrightarrow l'\}}$$

$$\text{(R-Read)}\quad \frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle \;\longmapsto\; l :: P\sigma \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle}$$

$$\text{(R-New)}\quad l :: \mathbf{new}(l').P \;\longmapsto\; (\nu l')(l :: P \parallel l' :: \mathbf{nil})$$

$$\text{(R-Par)}\quad \frac{N_1 \;\longmapsto\; N_1'}{N_1 \parallel N_2 \;\longmapsto\; N_1' \parallel N_2}$$

$$\text{(R-Res)}\quad \frac{N \;\longmapsto\; N'}{(\nu l)N \;\longmapsto\; (\nu l)N'}$$

$$\text{(R-Struct)}\quad \frac{N \equiv M \quad M \;\longmapsto\; M' \quad M' \equiv N'}{N \;\longmapsto\; N'}$$

Table 3. tKlaim Reduction Relation

Intuitively, a tuple matches a template if they have the same number of fields, and corresponding fields match. Formally, function *match* is defined in Table 2, where we let 'ϵ' to be the empty substitution and '$\circ$' to denote substitutions composition. Here, a substitution σ is a mapping of localities and basic values for variables; $P\sigma$ denotes the (capture avoiding) application of σ to P.

The intuition beyond the operational rules of tKlaim now follows. Rule (R-Out) evaluates the expressions within the argument tuple and sends the resulting tuple to the target node. However, this is possible only if the source and the target nodes are directly connected. Rule (R-Eval) is similar: a process can be spawned at l' by a process running at l only if l and l' are directly connected. Rules (R-In) and (R-Read) require existence of a matching datum in the target node and a connection between the source and the target node. The tuple is then used to replace the free occurrences of the variables bound by the template in the continuation of the process performing the actions. With action **in** the matched datum is consumed while with action **read** it is not. Rule (R-New) says that execution of action **new**(l') simply adds a restriction over l' to the net; from then on, a new node with locality l' and its links with other nodes of the net can be allocated/deallocated by using law (RNode). Rules (R-Par), (R-Res) and (R-Struct) are standard.

tKlaim adopts a Linda-like [16] communication mechanism: data are anonymous and associatively accessed via pattern matching, and communication is asynchronous.

Indeed, even if there exist prefixes for placing data to (possibly remote) nodes, no synchronization takes place between (sending and receiving) processes.

2.3 Observational Semantics

We now present a preorder on tKLAIM nets yielding sensible semantic theories. We follow the approach put forward in [12] and use *may testing* preorder and the associated equivalence. Intuitively, two nets are may testing equivalent if they cannot be distinguished by any external observer. More precisely, an *observer O* is a net containing a node whose address is a reserved locality name `test`. A computation reports *success* if, along its execution, a datum at node `test` appears; this is written $\stackrel{OK}{\Longrightarrow}$.

Definition 2 (May Testing Preorder and Equivalence). May testing preorder, $\sqsubseteq$, *is the least preorder on* tKLAIM *nets such that, for every* $N \sqsubseteq M$, *it holds that* $N \parallel O \stackrel{OK}{\Longrightarrow}$ *implies* $M \parallel O \stackrel{OK}{\Longrightarrow}$, *for any observer O.*

May testing equivalence, $\simeq$, *is defined as the intersection of* $\sqsubseteq$ *and* $\sqsupseteq$.

3 Implementing Distant Communications: A Routing Messenger

To better clarify the features of our calculus, we now present a simple routing application. In the setting we introduced, a process at l can perform action $\mathbf{out}(t)@l'$ only if l and l' are directly connected. We now supply a protocol to deliver t from l to l' under the assumption that there exists a path of links from l to l' in the connection graph. Clearly, the example we present can be readily adapted to implement distant **in**, **read** and **eval** actions.

For the sake of readability, we define a conditional construct to select one between two processes for execution while discarding the other. It is defined as:

$$\mathbf{if}\ \ell = \ell'\ \mathbf{then}\ P\ \mathbf{else}\ Q \triangleq \mathbf{new}(l).\mathbf{out}(\ell = \ell')@l.(\ \mathbf{in}(\mathtt{tt})@l.P \mid \mathbf{in}(\mathtt{ff})@l.Q\ )$$

where we assume `tt` and `ff` to be boolean values, and '=' be the equality test for names. As intended, it can be easily proved that, if $\ell = \ell'$, then only P can evolve and, if $\ell \neq \ell'$, then only Q can evolve (see Proposition 2 below).

We assume that, for each pair of (possibly indirectly) connected localities l_1 and l_2, there is a (permanent and unique) tuple $\langle l_2, l_3\rangle$ at l_1 recording the next directly connected node l_3 to visit for reaching l_2.[1] Now, the mobile agent delivering datum t from l to l' is

$$\begin{array}{rcl} \mathit{Deliver}(t,l,l') & \triangleq & \mathbf{new}(l'').\mathbf{out}(l)@l''.\mathbf{rec}\,X.\mathbf{in}(!x)@l''.\mathbf{read}(l', !y)@x. \\ & & \mathbf{if}\ y = l'\ \mathbf{then}\ \mathbf{out}(t)@l'\ \ \mathbf{else}\ \mathbf{out}(y)@l''.\mathbf{eval}(X)@y \end{array}$$

Intuitively, the restricted locality l'' acts as a repository storing the locality where the process is currently running. The recursive part first retrieves the current locality x, then

[1] The main goal of *routing algorithms* is to build this data structure (called *routing table*) at the outset and to maintain its consistency during net evolution. In our setting, links do not change; hence, the routing table is calculated once and for all at the outset. For a more dynamic setting, see Sections 4.2 and 5.

gets the next node y to visit before reaching l'; if such a node is l' itself, then the current node is directly connected to l' and action $\mathbf{out}(t)@l'$ ends the process, otherwise the process migrates to node y and iterates its behaviour.

Soundness of the protocol can be formalized as follows. Let l' be the address of a node in N; if l is connected to l' in N, then

$$N \parallel l :: Deliver(t, l, l') \simeq N \parallel l' :: \langle t \rangle \quad (1)$$

otherwise

$$N \parallel l :: Deliver(t, l, l') \simeq N \parallel l :: \mathbf{nil} \quad (2)$$

Intuitively, Equation (1) states that, if the target node is reachable from the source one, then agent $Deliver$ properly forwards the message to its destination; moreover, the agent has no other visible effect on the overall net behaviour. On the other hand, if the source and the target nodes are not (even indirectly) connected, then the activity of agent $Deliver$ is completely transparent for any external observer.

Proof

To prove the above equations, we first give a Proposition collecting some very simple equational laws. Then, we give another simple Proposition establishing soundness of the if-then-else construct introduced before. The proofs of these facts can be easily carried on by exploiting a co-inductive (bisimulation-based) proof technique provided in Appendix A.

Proposition 1.

1. $l :: \mathbf{out}(t)@l'.P \parallel \{l \leftrightarrow l'\} \simeq l :: P \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle$
2. $l :: \mathbf{eval}(\ )@l'.P \parallel \{l \leftrightarrow l'\} \simeq l :: P \parallel \{l \leftrightarrow l'\} \parallel l' ::$
3. $(\nu l')(l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle) \simeq (\nu l')(l :: P\sigma)$ if $act\ (\mathcal{E}[\![T]\!]; t) = \sigma$
4. $(\nu l)(l :: C) \simeq \mathbf{0}$ whenever C is a datum $\langle t \rangle$, a stuck process nil or the parallel composition of such components
5. $l :: \mathbf{new}(l').P \simeq (\nu l')(l :: P)$

Proposition 2.

1. if $l = l$ then P else $\simeq P$
2. if $l = l'$ then P else $\simeq$, whenever $l \neq l'$.

Finally, we need a Proposition that regulates the access to the routing tables stored in the nodes of the routing messenger example.

Proposition 3. If observers do not provide data of the form $\langle l', \cdot \rangle$ located at l, never emit data of the form $\langle l', \cdot \rangle$ at l, and never remove datum $\langle l', l'' \rangle$ from l, then

1. $l :: \langle l', l'' \rangle \mid \mathbf{read}(l', !x)@l.P \simeq l :: \langle l', l'' \rangle \mid P[l''/x]$
2. $l :: \mathbf{read}(l', !x)@l.P \simeq l :: \mathbf{nil}$

Notice that considering observers like those required in Proposition 3 is natural, if we consider data $\langle l', \cdot \rangle$ as entries of the routing table of l. No entry can be ever added/removed during the computation: the routing table is deterministic (there is at

most one path for each l), it is calculated at the outset and never changes during the computation.

We now prove the soundness of the protocol. For Equation (1), we know that, if l and l' are connected, then there is a path $l \; l_0 \; l_1 \; \ldots \; l_n \; l'$ (for $n \geq 0$) in the connection graph underlying N. We now proceed by induction on n.

Base. In this case, $l' = l$ and hence

$$N \;\; l :: \mathrm{Deliver}(t, l, l) \;\; (\nu\, l)(N \;\; l :: \mathrm{if}\ l = l\ \mathrm{then}\ \mathrm{out}(t)@l \;\; \mathrm{else}\ \mathrm{out}(l)@l \;\; .\mathrm{eval}(\mathrm{rec}\, X.\mathrm{in}(!x)@l \;\; .\mathrm{read}(l, !y)@x.\ \mathrm{if}\ y = l\ \mathrm{then}\ \mathrm{out}(t)@l\ \mathrm{else}\ \mathrm{out}(y)@l \;\; .\mathrm{eval}(X)@y)@l) \;\; N \;\; l :: t$$

The first equality is proved by using Propositions 1.5/.1/.3 and 3.1, and by the fact that structurally equivalent nets are also may testing equivalent (easy to prove). The second equality relies on Propositions 2.1, 1.1 and 1.4.

Induction. Let $l \; l_0 \; l_1 \; \ldots \; l_n \; l'$. Thus

$$N \;\; l :: \mathrm{Deliver}(t, l, l') \;\; (\nu\, l)(N \;\; l :: \mathrm{if}\ l_1 = l\ \mathrm{then}\ \mathrm{out}(t)@l \;\; \mathrm{else}\ \mathrm{out}(l_1)@l \;\; .\mathrm{eval}(\mathrm{rec}\, X.\mathrm{in}(!x)@l \;\; .\mathrm{read}(l, !y)@x.\ \mathrm{if}\ y = l\ \mathrm{then}\ \mathrm{out}(t)@l\ \mathrm{else}\ \mathrm{out}(y)@l \;\; .\mathrm{eval}(X)@y)@l_1)$$

$$N \;\; l :: t \quad \text{if } l_1 = l \qquad N \;\; l_1 :: \mathrm{Deliver}(t, l_1, l') \quad \text{otherwise}$$

$$N \;\; l :: t$$

The first and the second equalities when $l_1 = l'$ (thus $n = 1$) are proved like in the base case. The second equality when $l_1 \; l'$ is proved by using Propositions 2.2 and 1.1/.2/.1/.5. The third equality relies on a straightforward induction or by using reflexivity of $\equiv$, according to whether $l_1 \; l'$ or not.

We are left with Equation (2): l and l' are not connected. Thus, there is no association $l', \cdot$ in the routing table of l and it will never appear. Thus,

$$N \;\; l :: \mathrm{Deliver}(t, l, l') \equiv (\nu\, l)(N \;\; l :: \mathrm{read}(l', !y)@l.\mathrm{if}\ y = l \;\; \ldots) \;\; N \;\; l :: \mathrm{nil}$$

The first equivalence is proved like above, while the second one relies on Propositions 3.2 and 1.4.

4 Modelling Failures

We now enrich the basic framework with a mechanism for modelling various forms of failures, a key feature of global computers. We start with failure of nodes and node components; then, we use this setting to prove the properties of a distributed fault-tolerant protocol. Finally, we sketch a minor modification of our framework to also let node links fail.

4.1 Failure of Nodes and Node Components

We start by letting only nodes and node components fail. This is simply achieved by adding the operational rule

$$\text{(R-FailN)} \qquad l :: C \longmapsto \mathbf{0}$$

that models corruption of data (*message omission*) if $C \triangleq \langle t_1 \rangle | \ldots | \langle t_n \rangle$, node (*fail-silent*) failure if $l :: C$ collects all the clones of l, and abnormal termination of some processes running at l otherwise. Modelling failures as disappearance of a resource (a datum, a process or a whole node) is a simple, but realistic, way of representing failures, specifically fail-silent and message omission, in a global computing scenario [4]. Indeed, while the presence of data/nodes can be ascertained, their absence cannot because there is no practical upper bound to communication delays. Thus, failures cannot be distinguished from long delays and should be modelled as totally asynchronous and undetectable events.

For the sake of clarity, we shall denote with $\sqsubseteq_f$ and $\simeq_f$ the may testing preorder and equivalence obtained when adding rule (R-FailN) to the rules in Table 3.

A Distributed Fault-tolerant Protocol: k–set Agreement We now use may testing to verify the correctness of *k–set agreement* [8], a simple distributed fault-tolerant protocol. Suppose to have an asynchronous message-passing totally-connected distributed system with n principals; each principal has an input value (taken from a totally ordered set) and must produce an output value. The principals can fail and we adopt a fail-silent model of failures; however, the communication medium is reliable, i.e. messages sent will surely be received although the order and the moment in which messages will arrive are unpredictable because of asynchrony. The *agreement* problem requires to find a protocol that satisfies three properties: *termination* (i.e. the non-faulty principals eventually produce an output), *agreement* (i.e. all the non-faulty principals produce the *same* output value) and *validity* (i.e. the output value must be one of the input values). It is well-known (see, e.g. [2]) that a solution for this problem does not exists even if a single failure occurs.

The *k–set agreement* problem relaxes the agreement property to enable the existence of a solution. Indeed, for each $1 \leq k \leq n$, it requires that, assuming at most $k-1$ faulty principals, the non-failed principals successfully complete their execution by producing outputs taken from a set whose size is at most k. Notice that for $k = 1$ we get the agreement problem without failures.

A possible solution for the k–set agreement problem is given by the following protocol, taken from [2], executed by each principal:

(i) send your input value to all principals (including yourself)
(ii) wait to receive $n - k + 1$ values
(iii) output the minimum value received

In this way, if we call $\mathcal{I}$ the set of the input values, the set of output values $\mathcal{O}$ is formed by the k smallest values in $\mathcal{I}$ (for the sake of simplicity, we assume that the elements in $\mathcal{I}$ are pairwise distinct; however, the protocol works even if input values are duplicated – in this case $\mathcal{I}$ and $\mathcal{O}$ are multisets).

We let integers play the role of the input/output values, while principals are represented as distinct nodes, whose addresses are taken from the set $\widetilde{l} \triangleq \{l_1, \ldots, l_n\}$; moreover, we let $d_i \in \mathcal{I}$ to be the input value of the principal associated to the node whose address is l_i. Once we fix the value for k, node l_i hosts the process

$$P_i^k \;\triangleq\; \mathbf{out}(d_i)@l_1.\ \ldots\ .\mathbf{out}(d_i)@l_n.\mathbf{in}(!z_1^i)@l_i.\ \ldots\ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l$$

with $m_i \triangleq min\{z_j^i : j = 1, \ldots, n-k+1\}$ and l be a distinct locality used to collect output values. The net implementing the whole protocol is

$$N_n^k \;\triangleq\; (\nu\widetilde{l})\Big(\prod_{i=1}^{n}\; l_i :: P_i^k\Big)$$

where we restricted the localities associated to the principals because no external context is allowed to interfere with the execution of the protocol. Notice that, having restricted the $\widetilde{l}$, all the principals are connected and no **out** prefix will ever block P_i^k (because of law (RNode)). However, this does not prevent failures: the failure of (a reduct of) P_i^k is indeed the failure of principal i.

A formulation of the three properties for the k–set agreement problem is given by Equations (3) and (4) below, whose proof is in Appendix B. The formalization of k–set agreement and validity properties is given by the Equation

$$N_n^k \;\simeq_f\; M_n^k \tag{3}$$

There, we exploit the auxiliary net

$$M_n^k \;\triangleq\; (\nu\widetilde{l},\widetilde{l'})\Big(\prod_{i=1}^{n}\; (\, l_i :: Q_i^k \;\|\; l'_i :: \prod_{w\in O}\langle w\rangle\,)\Big)$$

where

$$\begin{aligned} Q_i^k \;\triangleq\;\; & \mathbf{out}(d_i)@l_1.\ \cdots\ .\mathbf{out}(d_i)@l_n.\mathbf{in}(!z_1^i)@l_i.\cdots. \\ & \mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{in}(m_i)@l'_i.\mathbf{out}(m_i)@l \end{aligned}$$

We assume that nodes whose addresses are in $\widetilde{l'}$ cannot fail; this is reasonable because they are only auxiliary nodes and hence their failure is irrelevant for the original formulation of the problem. Intuitively, node l'_i acts as a repository for l_i and contains the possible output values (i.e. the elements of O), while the last **in** action of Q_i^k is a test for checking that the output value produced by the principal i is in O. The net M_n^k obviously satisfies the wanted properties since its principals output only values present in O. The fact that $|O| = k$ then implies the k–set agreement property, while the fact that $O \subseteq \mathcal{I}$ implies validity.

In order to prove the termination property, it suffices to prove that

$$l :: \prod_{j=1}^{n-k+1}\langle\rangle \quad \sqsubseteq_f \quad \hat{N}_n^k \tag{4}$$

where $\hat{N}_n^k \;\triangleq\; (\nu\widetilde{l})\Big(\prod_{i=1}^{n}\; (l_i :: \hat{P}_i^k \;\|\; \{l_i \leftrightarrow l\})\Big)$ and processes $\hat{P}_i^k$ is defined like P_i^k with action $\mathbf{out}()@l$ in place of $\mathbf{out}(m_i)@l$. Clearly, if we only consider termination, N_n^k and $\hat{N}_n^k$ are equivalent, in the sense that a non-faulty principal produces an output value in the first net if and only if its counterpart produces an output in the second net. Equation (4) implies termination of the protocol, since it requires that at least $n-k+1$

tuples are produced at l; by definition of the protocol, this is possible only if $n - k + 1$ principals terminate successfully.

In conclusion, we want to remark that other solutions to the agreement problem in presence of failures have been given in literature. Some of these solutions use *failure detectors* [7, 2]. Recently, one such solution has been formalized and proved sound by using a process algebraic approach [15]. The solution in *loc.cit.* is, however, heavier than ours and exploits properties of the operational semantics, instead of working in a (simpler) equational setting. Moreover, it exploits failure detectors which are hardly implementable in a global computing scenario.

4.2 Failure of Inter-Node Connections

The philosophy underlying our failure model can be easily adapted to deal with link failures too. To this aim, we only need to add the operational rule

$$\text{(R-FailC)} \quad \{l_1 \leftrightarrow l_2\} \longmapsto \mathbf{0}$$

that models the (asynchronous and undetectable) failure of the link between nodes l_1 and l_2.

Discovering Neighbours. Since the (multi)set of links in a net can change during computations, the framework presented in Section 3 needs some adaption. Indeed, like in practice, routing tables calculated at the outset must be updated during a computation, because the original topology can change at runtime. This task is usually carried on by *routing algorithms*. Several proposals have been presented in literature and different standards use different solutions.

In general, routing algorithms are repeated at regular time intervals and consist in two main phases: first, each node discovers its neighbours; then, it calculates its routing table by usually sharing local information with its neighbours. We present here a simple way to implement in tKlaim the first phase; the (more challenging) study of the second phase is left for future work.

Neighbours can be discovered in a simple way. Each node l can try to send a "hallo" message to another node l'; if this action succeeds, then a connection between l and l' does exist; otherwise, nothing can be said (e.g., the message could get lost or the link could be congested and this caused a delay to the message). In our framework, no explicit message is needed: a simple action $\mathbf{eval}(\mathbf{nil})@l'$ performed at l can be used as test for existence of link $\{l \leftrightarrow l'\}$ in the net. By letting $\simeq_f$ still denote the may testing equivalence in this refined framework, soundness of our solution can be formalized and proved via the following equality (whose proof can be easily carried on by exploiting the proof technique in Appendix A):

If $N \equiv N' \parallel \{l \leftrightarrow l'\}$, for some N', then

$$N \parallel l :: \mathbf{eval}(\mathbf{nil})@l'.\mathbf{out}(\text{“}CONN\text{”}, l, l')@l \;\simeq_f\; N \parallel l :: \langle \text{“}CONN\text{”}, l, l' \rangle$$

5 Modelling Dynamic Connections

Finally, we present another variation of the basic language that let connections dynamically evolve. To this aim, we add two actions to create and destroy a link, respectively; formally, we add the production

$$a \ ::= \ \ldots \ \mid \ \mathbf{login}(\ell) \ \mid \ \mathbf{logout}(\ell)$$

to the syntax of Table 1. Intuitively, the first action, when executed at node l, creates a new link between l and ℓ, if the latter name is associated to a network node. Conversely, the second action, when executed at node l, dissolves a link between l and ℓ, if such a link exists. These intuitions are formalised by the following operational rules, that must be added to those in Table 3:

$$\text{(R-Login)} \qquad l :: \mathbf{login}(l').P \parallel l' :: \mathbf{nil} \ \longmapsto \ l :: P \parallel \{l \leftrightarrow l'\}$$

$$\text{(R-Logout)} \qquad l :: \mathbf{logout}(l').P \parallel \{l \leftrightarrow l'\} \ \longmapsto \ l :: P \parallel l' :: \mathbf{nil}$$

Again, for the sake of clarity, we denote with $\simeq_d$ the may testing equivalence in the calculus with dynamic connections.

Message Delivering in a Dynamic Net. To conclude, we now give an application of our theory in a setting where node links change dynamically. To this aim, we use a simplified scenario inspired by the *handover protocol*, proposed by the European Telecommunication Standards Institute (ETSI) for the GSM Public Land Mobile Network (PLMN). The formal specification of the protocol and its service specification are in [18]; we use here an adaption of their approach.

The PLMN is a cellular system which consists of Mobile Stations (MSs), Base Stations (BSs) and Mobile Switching Centers (MSCs). MSs are mobile devices that provide services to end users. BSs manage the interface between the MSs and a stationary net; they control the communications within a geographical area (a cell). Any MSC handles a set of BSs; it communicates with them and with other MSCs using a stationary net.

A new user can enter the system by connecting its MS with a MSC that, in turn, will decide the proper BS responsible for such a MS. Then, messages sent from the user are routed to their destinations by the BS, passing through the MSC handling the BS. However, it may happen that the BS responsible for a MS should be changed during the computation (e.g., because the MS left the area associated to the BS and entered in the area associated to a different BS). In this case, the MSC should carry on the rearrangements needed to cope with the new situation, without affecting the end-to-end communication.

We now model the key features of a PLMN in tKlaim; however, for the sake of simplicity, several aspects will be omitted, like, e.g., the criterion to choose a proper BS for a given MS, or the event originating an handover. Both MSs, BSs and MSCs are modelled as nodes. For the sake of simplicity, we consider a very simple PLMN, with one MSC (whose address is $\mathtt{M}$) and two BSs (whose addresses are $\mathtt{B}_1$ and $\mathtt{B}_2$, resp.).

Let us start witht the process that performs the connecting formalities in $\mathtt{M}$.

$$\begin{aligned} \mathit{ENTER} \ \triangleq \ & < \textit{gather a new connection from } l > .\mathbf{read}(!B)@\mathtt{BSlist}. \\ & \mathbf{eval}(\mathbf{login}(l))@B.\mathbf{logout}(l).\mathbf{out}(l, B)@\mathtt{Table} \end{aligned}$$

When a new user want to enter the PLMN, it has to perform a **login**(M) from his MS, whose address is l; this generates an interrupt in M (that we do not model here) by which the MSC can gather the address of the MS. This address, together with other information (like the geographical area of the user or its credentials), are used by the MSC to choose a proper BS; in our simplified framework, we let M take a BS's address from a private repository BSlist. Then, the MSC creates a new link from the chosen BS to the MS and destroys the link from itself to the MS. Finally, it records in a private repository Table the fact that the new MS is under the control of the chosen BS.

Once entered the PLMN, the new user can send some data d to (the MS of) a remote user (whose address is l'); this is achieved by letting his MS (whose address is l) perform an action of the form **out**('send', l', d)@l. Then, the BSs associated to l and l' come into the picture to properly deliver the message. In particular, let $\mathtt{B}_i$ be the BS associated to l and $\mathtt{B}_j$ be the BS associated to l' (for $i, j \in \{1, 2\}$). Then, the message is forwarded from $\mathtt{B}_i$ to $\mathtt{B}_j$ by the process

$$FWD_i \triangleq \mathbf{read}(!x, \mathtt{B}_i)@\mathtt{Table}.\mathbf{in}(\text{'send'}, !y, !z)@x.\mathbf{in}(y, !B)@\mathtt{Table}.\mathbf{out}(y, z)@B$$

This process first retrieves the address of a MS associated to $\mathtt{B}_i$ (in particular, l); then, it collects the message and forwards it to the BS associated to the destination MS. Notice that, in doing this, it 'locks' the link between l' and $\mathtt{B}_j$ until the message will be delivered to l' (see below); this is necessary to avoid that a handover may interfere with the message delivering. Then, the message is collected by $\mathtt{B}_j$ and passed to l' by the process

$$CLT_j \triangleq \mathbf{in}(!dest, !mess)@\mathtt{B}_j.\mathbf{out}(mess)@dest.\mathbf{out}(dest, \mathtt{B}_j)@\mathtt{Table}$$

This process retrieves the message sent by $\mathtt{B}_i$ and passes it to the final MS; then, it releases the 'lock' on the link $\{\mathtt{B}_j \leftrightarrow l'\}$ acquired by $\mathtt{B}_i$ by putting back in Table the tuple $\langle l', \mathtt{B}_j \rangle$. Clearly, there are also processes FWD_j and CLT_i running in $\mathtt{B}_j$ and $\mathtt{B}_i$ respectively, but they do not play any role here.

Finally, the handover is handled by the MSC via the following process:

$$\begin{aligned} HNDVR \triangleq\ & \mathbf{in}(!x, !B)@\mathtt{Table}.\mathbf{read}(!B')@\mathtt{BSlist}. \\ & \mathbf{eval}(\mathbf{logout}(x))@B.\mathbf{eval}(\mathbf{login}(x))@B'.\mathbf{out}(x, B')@\mathtt{Table} \end{aligned}$$

This process first selects a MS-to-BS association to be changed (the reason why this is needed is not modelled here); then, it chooses a new BS, properly changes the links between the MS and the BSs, and updates the repository Table.

The overall resulting system is

$$\begin{aligned} SYS \triangleq\ & (\nu \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\mathtt{M} :: *ENTER \mid *HNDVR \\ & \parallel \mathtt{BSlist} :: \langle \mathtt{B}_1 \rangle \mid \langle \mathtt{B}_2 \rangle \parallel \mathtt{Table} :: \mathbf{nil} \\ & \parallel \mathtt{B}_1 :: *FWD_1 \mid *CLT_1 \parallel \mathtt{B}_2 :: *FWD_2 \mid *CLT_2) \end{aligned}$$

where $*P$ denotes the replication of P and stands for an unbounded number of copies of P running in parallel. Replication can be easily encoded through recursion by letting $*P$ be a shortcut for **rec** $X.(P|X)$. Soundness of the system can be formulated as (a proof

sketch is in Appendix C):

$$\begin{array}{l}(\nu l)(l :: \mathbf{login}(\mathtt{M}).\mathbf{out}(\text{`send'}, l', \text{`HI'})@l \parallel l' :: \mathbf{login}(\mathtt{M}) \parallel S\,YS\,) \\ \quad \simeq_d \quad (\nu l)(l' :: \langle \text{`HI'} \rangle \parallel S\,YS\,)\end{array} \tag{5}$$

Notice that l is restricted only to simplify proofs: soundness of the protocol is not affected by the fact that the MSs are public or not.

6 Related Work

In the last decade, several languages for global computers have been proposed in literature; we mention here only the most strictly related ones.

In DJoin [14], located mobile processes are hierarchically structured and form a tree-like structure evolving during the computation. Entire subtrees, and not only single processes, can move and fail. Communication takes place in two steps: first, the sending process sends a message on a channel; then, the ether (i.e. the environment containing all the nodes) delivers the message to the (unique) process that can receive on that channel. Failures are programmed (i.e., they result from the execution of some process actions) and can be detected by processes. We believe that the setting presented in this paper is more realistic than DJoin: first, we consider interconnection topologies that are more general than trees; second, we do not assume any implicit engine for distant communications; third, we model failures in a way that is closer to actual global computers.

The Ambient calculus [5] is an elegant notation to model hierarchically structured distributed applications. Like our work, the calculus is centered around the notion of connections between ambients, that are containers of processes and data. Each language primitive can be executed only if the ambient hierarchy is structured in a precise way; e.g., an ambient n can enter an ambient m only if n and m are sibling, i.e. they are both contained in the same ambient. However, like DJoin, Ambient strongly relies on a tree-like structure for the ambient hierarchy. Moreover, to the best of our knowledge, no explicit notion of failures, close to actual global computing requirements, has been ever given for Ambient.

[20] presents Nomadic Pict, a distributed and agent-based language based on the π-calculus. It relies on a flat net where named agents can roam. Communication between two agents can take place only if they are located at the same node (thus no low-level remote communication is allowed). However, the language also provides a (high-level) primitive for remote communication, that transparently delivers a message to an agent even if the latter is not co-located with the sender. This primitive is then encoded in the low-level calculus by a central forwarding server, implemented by only using the low-level primitives. The assumption that only co-located agents can communicate is, in our opinion, too stringent. Moreover, it is not clear to us how the theory can be adapted when failures enter the picture.

Finally, we want to remark that the use of observational equivalences to state and proof soundness of protocols is a well-established technique in the field of process calculi; some examples are [1, 17, 18, 21]. In particular, in the last paper, an automatic verification tool to prove equivalences in the π-calculus is described. As an application,

the authors automatically verify an equality (more involved than ours) stating soundness of the PLMN example.

References

1. M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. In *Proc. of CONCUR'97*, volume 1243 of *LNCS*, pages 59–73. Springer, 1997.
2. H. Attiya and J. Welch. *Distributed Computing*. McGraw Hill, 1998.
3. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th WETICE*, pages 110–115. IEEE, 1998.
4. L. Cardelli. Abstractions for mobile computation. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 51–94. Springer, 1999.
5. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
6. S. Castellani, P. Ciancarini, and D. Rossi. The ShaPE of ShaDE: a coordination system. Tech. Rep. UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. di Bologna, Italy, 1996.
7. T. Chandra and S.Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.
8. S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132–158, 1993.
9. N. Davies, S. Wade, A. Friday, and G. Blair. L^2imbo: a tuple space based platform for adaptive mobile applications. In *Int. Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP'97)*, 1997.
10. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
11. R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. Technical Report 07/2004, Dip. di Informatica, Univ. di Roma "La Sapienza". Available at `http://www.dsi.uniroma1.it/~gorla/papers/bo4k-full.pdf`.
12. R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
13. D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proc. of ISADS 2001*, pages 278–286. IEEE, 2001.
14. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
15. R. Fuzzati, M. Merro, and U. Nestmann. Modelling Consensus in a Process Calculus. In *Proc. of CONCUR'03*, volume 2761 of *LNCS*. Springer-Verlag, 2003.
16. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
17. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.
18. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Formal Aspects of Computing*, 4:497–543, 1992.
19. J. Parrow. An introduction to the pi-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.
20. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proc. of POPL'01*, pages 116–127. ACM Press, 2001.
21. B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In *Proc. of CAV '94*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.

A Labelled Bisimulation as a Proof Technique for May Testing Equivalence

Here we briefly sum up and re-adapt some of the theory presented in [11]. The main goal of this section is to provide a easy to handle way to establish equalities under may testing equivalence. Indeed, may testing is hardly usable in practice because of its universal quantification over observers. To this aim, we present a co-inductive bisimulation that can be used to infer equalities under a may testing semantics.

The first step in this direction is to make apparent the actions a net intend to perform in order to evolve. To this aim, we define a *labelled transition relation*, $\xrightarrow{\alpha}$, defined as the least relation over nets induced by the inference rules in Table 4. Transition labels take the form

$$\chi ::= \tau \mid (\nu\widetilde{l})\, I \mathbin{@} l \mid l_1 \rightarrow l_2 \qquad \alpha ::= \chi \mid l_1 : \triangleright\, l_2 \mid l_1 : t \triangleleft l_2$$

where

$$I ::= \mathbf{nil} \mid (\nu\widetilde{l})\, \langle t \rangle \mathbin{@} l$$

groups together those node components, called *inert*, that cannot perform any action. We will write $bn(\alpha)$ for $\widetilde{l}$ if $\alpha = (\nu\widetilde{l})\, I \mathbin{@} l$, and for $\emptyset$ otherwise; $fn(\alpha)$ is defined accordingly. Transitions labelled with τ are the only computational steps: it can be proved that they are in a 1-to-1 correspondence with reductions. Labels $\mathbf{nil} \mathbin{@} l$, $(\nu\widetilde{l})\, \langle t \rangle \mathbin{@} l$ and $l_1 \rightarrow l_2$ describe net's structure: they signal, respectively, the existence of a node with address l, of a datum $\langle t \rangle$ at node l containing the restricted names $\widetilde{l}$, and of a link between l_1 and l_2. Finally, label $l_1 : \triangleright\, l_2$ ($l_1 : t \triangleleft l_2$, resp.) declares the intention of some process located at l_1 to send components to (receive the datum t from) node l_2.

Let us now briefly comment on some rules of the LTS; most of them are adapted from the π-calculus [19]. Rule (LTS-Exists) points out existence of nodes (label $\mathbf{nil} \mathbin{@} l$) or of data (label $\langle t \rangle \mathbin{@} l$), while rule (LTS-Link) points out existence of links. Rules (LTS-Out) and (LTS-Eval) spawn a component, but they require the existence of a link between the source and the target node in order to get concretised, see rule (LTS-Send). Rules (LTS-In) and (LTS-Read) require the existence of the chosen datum in the target node, and of a proper link, in order to get concretised, see rule (LTS-Comm). Rule (LTS-Open) signals extrusion of bound names. As in some presentations of the π-calculus, (LTS-Open) is used to investigate the capability of processes to export bound names, rather than to really extend the scope of bound names. To this last aim, structural scope extension is used; in fact, in rules (LTS-Send) and (LTS-Comm) labels do not carry any restriction on names (whose scope must have been previously extended). Rules (LTS-Res), (LTS-Par) and (LTS-Struct) are standard.

In [11], we have defined a *labelled bisimulation* on top of a similar LTS. There, we have also proved that such a relation is an equivalence and it is a sound proof technique for a similarly defined may testing equivalence[2]. As usual, we let $\Rightarrow$ stand for the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\overset{\alpha}{\Longrightarrow}$ stand for $\Rightarrow \xrightarrow{\alpha} \Rightarrow$ (where juxtaposition

[2] More precisely, in [11] we proved that a similar bisimulation coincides with a standardly defined barbed congruence. This fact, together with the usual inclusion of barbed congruence in may testing equivalence, yields the claimed proof technique.

(LTS-Out)
$$\frac{\mathcal{E}[\![\, t \,]\!] = t'}{l :: \mathbf{out}(t)@l'.P \xrightarrow{l :\triangleright\, l'} l :: P \parallel l' :: \langle t' \rangle \parallel \{l \leftrightarrow l'\}}$$

(LTS-New)
$$\frac{}{l :: \mathbf{new}(l').P \xrightarrow{\tau} (\nu l')(l :: P)}$$

(LTS-Eval)
$$\frac{}{l :: \mathbf{eval}(Q)@l'.P \xrightarrow{l :\triangleright\, l'} l :: P \parallel l' :: Q \parallel \{l \leftrightarrow l'\}}$$

(LTS-Exists)
$$\frac{}{l :: I \xrightarrow{I\ @\ l} l :: \mathbf{nil}}$$

(LTS-In)
$$\frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{in}(T)@l'.P \xrightarrow{l :\, t \,\triangleleft\, l'} l :: P\sigma \parallel \{l \leftrightarrow l'\}}$$

(LTS-Link)
$$\frac{}{\{l_1 \leftrightarrow l_2\} \xrightarrow{l_1 \rightarrow l_2} \mathbf{0}}$$

(LTS-Read)
$$\frac{match(\mathcal{E}[\![\, T \,]\!]; t) = \sigma}{l :: \mathbf{read}(T)@l'.P \xrightarrow{l :\, t \,\triangleleft\, l'} l :: P\sigma \parallel \{l \leftrightarrow l'\} \parallel l' :: \langle t \rangle}$$

(LTS-Comm)
$$\frac{N_1 \xrightarrow{l_1 :\, t \,\triangleleft\, l_2} N_1' \quad N_2 \xrightarrow{l_1 \rightarrow l_2} \xrightarrow{\langle t \rangle\ @\ l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

(LTS-Send)
$$\frac{N_1 \xrightarrow{l_1 :\triangleright\, l_2} N_1' \quad N_2 \xrightarrow{l_1 \rightarrow l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

(LTS-Open)
$$\frac{N \xrightarrow{(\nu \widetilde{l})\ \langle t \rangle\ @\ l'} N' \qquad l \in fn(t) - \{\widetilde{l}, l'\}}{(\nu l)N \xrightarrow{(\nu \widetilde{l}, l)\ \langle t \rangle\ @\ l'} N'}$$

(LTS-Res)
$$\frac{N \xrightarrow{\alpha} N' \qquad l \notin n(\alpha)}{(\nu l)N \xrightarrow{\alpha} (\nu l)N'}$$

(LTS-Par)
$$\frac{N_1 \xrightarrow{\alpha} N_2 \qquad bn(\alpha) \cap fn(N) = \emptyset}{N_1 \parallel N \xrightarrow{\alpha} N_2 \parallel N}$$

(LTS-Struct)
$$\frac{N \equiv M \xrightarrow{\alpha} M' \equiv N'}{N \xrightarrow{\alpha} N'}$$

Table 4. tKlaim Labelled Transition System

denotes relation composition); moreover, $\overset{\hat{\alpha}}{\Longrightarrow}$ stands for $\Rightarrow$, if $\alpha = \tau$, and for $\overset{\alpha}{\Longrightarrow}$, otherwise.

Definition 3. *A symmetric relation $\mathfrak{R}$ between* tKlaim *nets is a* (weak) bisimulation *if, for each $N_1\ \mathfrak{R}\ N_2$, it holds that:*

1. *if $N_1 \xrightarrow{\chi} N_1'$ then $N_2 \overset{\hat{\chi}}{\Longrightarrow} N_2'$ and $N_1'\ \mathfrak{R}\ N_2'$, for some N_2';*
2. *if $N_1 \xrightarrow{l_1 :\triangleright\, l_2} N_1'$ then $N_2 \parallel \{l_1 \leftrightarrow l_2\} \Rightarrow N_2'$ and $N_1'\ \mathfrak{R}\ N_2'$, for some N_2';*
3. *if $N_1 \xrightarrow{l_1 :\, t \,\triangleleft\, l_2} N_1'$ then $N_2 \parallel l_2 :: \langle t \rangle \parallel \{l_1 \leftrightarrow l_2\} \Rightarrow N_2'$ and $N_1'\ \mathfrak{R}\ N_2'$, for some N_2'.*

Bisimilarity, *$\approx$, is the largest bisimulation.*

Bisimilarity requires that τ-steps must be replied to with zero or more τ-steps. Similarly, labels of the form $(\nu \widetilde{l})\ I\ @\ l$ or $l_1 \rightarrow l_2$ must be replied to with the same label

(possibly together with some additional τ-step). This is necessary since such labels describe the structure of the net (its nodes, data and connections): to be equivalent, two nets must have at least the same structure.

Labels of kind χ describe facts, i.e. computational steps or the structure of a net; thus, they must be faithfully replied to. On the other hand, labels different from χ only express intentions; thus, they are handled differently. For example, the intention of sending a component, say $N_1 \xrightarrow{l_1 : \triangleright\, l_2} N_1'$, can be simulated by a net N_2 (in a context where l_1 and l_2 are connected) through execution of some τ-steps that lead to some N_2' equivalent to N_1'. Indeed, since we want our bisimulation to be a congruence, a context that provides a link between the source and the target nodes of the sending action must not tell apart N_1 and N_2. Similar considerations also hold for the case of the input actions (third item of Definition 3), but the context now is $[\cdot] \parallel \{l_1 \leftrightarrow l_2\} \parallel l_2 :: \langle t \rangle$.

Theorem 1. $\approx \;\subset\; \simeq$

Bisimulations with Failures. We only need to properly extend the LTS to model failures of nodes, node components and inter-node connections. This can be achieved with the expected rules:

$$\text{(LTS-FailN)} \quad l :: C \xrightarrow{\tau} \mathbf{0} \qquad\qquad \text{(LTS-FailC)} \quad \{l_1 \leftrightarrow l_2\} \xrightarrow{\tau} \mathbf{0}$$

The bisimulation defined on top of the modified LTS, that we write $\approx_f$, is a sound proof technique for the corresponding may testing equivalence $\simeq_f$.

Bisimulations with Dynamic Connections. First, we need to properly extend the LTS to accomodate actions **login** and **logout**. This can be achieved with the following rules:

$$\text{(LTS-Login)} \quad l_1 :: \mathbf{login}(l_2).P \xrightarrow{l_1 : l_2} l_1 :: P \parallel \{l_1 \leftrightarrow l_2\}$$
$$\text{(LTS-Logout)} \quad l_1 :: \mathbf{logout}(l_2).P \xrightarrow{l_1 : \neg\, l_2} l_1 :: P \parallel l_2 :: \mathbf{nil}$$

Clearly, $fn(\, l_1 \,:\, l_2\,) = fn(\, l_1 \,:\, \neg\, l_2\,) = \{l_1, l_2\}$ and $bn(\, l_1 \,:\, l_2\,) = bn(\, l_1 \,:\, \neg\, l_2\,) = \emptyset$. As we said in Section 5, the intention of connecting or disconnecting to a remote node (as expressed by rules (LTS-Login) and (LTS-Logout), respectively) gets concretised when the target node or the connection we want to delete do exist. This is formalised by the following rules:

$$\text{(LTS-Conn)} \quad \frac{N_1 \xrightarrow{l_1 : l_2} N_1' \qquad N_2 \xrightarrow{\mathbf{nil}\ @\ l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'} \qquad\qquad \text{(LTS-Disc)} \quad \frac{N_1 \xrightarrow{l_1 : \neg\, l_2} N_1' \qquad N_2 \xrightarrow{l_1 \rightarrow l_2} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'}$$

The definition of bisimulation extends $\approx$ to encompass also the intentions expressed by labels $l_1 \,:\, l_2$ and $l_1 \,:\, \neg\, l_2$. Formally, it suffices to extend Definition 3 with the following two requirements:

4. *if* $N_1 \xrightarrow{l_1 : l_2} N_1'$ *then* $N_2 \parallel l_2 :: \mathbf{nil} \Rightarrow N_2'$ *and* $N_1' \;\mathfrak{R}\; N_2'$, *for some* N_2';
5. *if* $N_1 \xrightarrow{l_1 : \neg\, l_2} N_1'$ *then* $N_2 \parallel \{l_1 \leftrightarrow l_2\} \Rightarrow N_2'$ *and* $N_1' \;\mathfrak{R}\; N_2'$, *for some* N_2'.

We call $\approx_d$ the resulting bisimilarity that is a sound proof technique for $\simeq_d$.

B Proofs for the k-Set Agreement Example

To prove the properties formulated in Section 4, we first need a new equality

$$l :: I_1 | \ldots | I_n \ \sqsubseteq_f \ l :: I_1 | \ldots | I_m \quad \text{if } n \leq m \qquad (\dagger)$$

Second, we need to smoothly adapt some of the equalities put forward by Proposition 1: the first equality holds only under the hypothesis that l' is restricted, while the third equality holds only under the further hypothesis that $\langle t \rangle$ is not corruptible at l' (with "$\langle t \rangle$ is not corruptible at l' ", we mean that $l' :: \langle t \rangle$ does never fail). Then, we prove Equation (3) as follows:

$$\begin{aligned}
N_n^k &\simeq_f (\widetilde{vl})\Big(\prod_{i=1}^{n} \ l_i :: \mathbf{in}(!z_1^i)@l_i. \ \ldots \ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}(m_i)@l \,|\, \langle d_1 \rangle \,|\, \ldots \,|\, \langle d_n \rangle \Big) \\
&\simeq_f (\widetilde{vl})\Big(\prod_{i=1}^{n} \ l_i :: \mathbf{out}(m'_i)@l \,|\, \langle d_{i_1} \rangle \,|\, \ldots \,|\, \langle d_{i_{k-1}} \rangle \Big) \\
&\simeq_f (\widetilde{vl}, \widetilde{l'})\Big(\prod_{i=1}^{n} \ (l_i :: \mathbf{in}(m'_i)@l'_i.\mathbf{out}(m'_i)@l \,|\, \langle d_{i_1} \rangle \,|\, \ldots \,|\, \langle d_{i_{k-1}} \rangle \ \parallel \ l'_i :: \prod_{w \in O} \langle w \rangle \,) \Big) \\
&\simeq_f (\widetilde{vl}, \widetilde{l'})\Big(\prod_{i=1}^{n} \ (l_i :: \mathbf{in}(!z_1^i)@l_i. \ \ldots \ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{in}(m_i)@l'_i.\mathbf{out}(m_i)@l \\
&\qquad\qquad\qquad\qquad |\, \langle d_1 \rangle \,|\, \ldots \,|\, \langle d_n \rangle \ \parallel \ l'_i :: \prod_{w \in O} \langle w \rangle \,) \Big) \\
&\simeq_f M_n^k
\end{aligned}$$

where m'_i denotes $m_i[\widetilde{d}/\widetilde{z}]$, with $\widetilde{d} \triangleq \{d_1, \ldots, d_n\} - \{d_{i_1}, \ldots, d_{i_{k-1}}\}$ and $\widetilde{z} \triangleq \{z_1, \ldots, z_{n-k+1}\}$. The first and the last steps have been inferred by applying several times (the revised formulation of) Proposition 1.1 . The second and the fourth steps have been inferred by applying several times (the revised formulation of) Proposition 1.3; notice that, since the number of failures is at most $k-1$, the number of non-corruptible data present in each l_i is at least $n-k+1$. The third step relies on Proposition 1.3 and .4 . It is worth to notice that $m'_i \in O$ because, since $|O| = k$, at least one principal whose input value, say d', is in O has not failed; hence d' has been received by all the (non-failed) principals. Moreover, we assumed that the $\widetilde{l'}$ cannot fail and hence the data they host are uncorruptable.

To conclude, we are left with proving Equation (4). This can be done very similarly as follows:

$$\begin{aligned}
\hat{N}_n^k &\simeq_f (\widetilde{vl})\Big(\prod_{i=1}^{n} \ l_i :: \mathbf{in}(!z_1^i)@l_i. \ \ldots \ .\mathbf{in}(!z_{n-k+1}^i)@l_i.\mathbf{out}()@l \,|\, \langle d_1 \rangle \,|\, \ldots \,|\, \langle d_n \rangle \parallel l :: \mathbf{nil} \Big) \\
&\simeq_f (\widetilde{vl})\Big(\prod_{i=1}^{n} \ l_i :: \mathbf{out}()@l \,|\, \langle d_{i_1} \rangle \,|\, \ldots \,|\, \langle d_{i_{k-1}} \rangle \parallel l :: \mathbf{nil} \Big) \\
&\simeq_f l :: \prod_{j=1}^{n} \langle\rangle \\
&\sqsupseteq_f l :: \prod_{j=1}^{n-k+1} \langle\rangle
\end{aligned}$$

The first two steps are derived in the same way. The third step is derived using (the revised version of) Proposition 1.1 and Proposition 1.4. The fourth step derives from (†).

C Proofs for the Message Delivering in a Dynamic Net

First, we need two laws for the primitives **login** and **logout**, that are quite expectable.

$$l :: \mathbf{login}(l').P \parallel l' :: \mathbf{nil} \;\simeq_d\; l :: P \parallel \{l \leftrightarrow l'\} \qquad (\star)$$

$$l :: \mathbf{logout}(l').P \parallel \{l \leftrightarrow l'\} \;\simeq_d\; l :: P \parallel l' :: \mathbf{nil} \qquad (\star\star)$$

Moreover, we also need an adapted version of Proposition 1.3 to deal with action **read**. It is defined as follows:

$$(\nu l')(\, l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle t \rangle\,) \;\simeq_d\; (\nu l')(l :: P\sigma \parallel l' :: \langle t \rangle) \quad \text{if } \mathit{match}(\mathcal{E}[\![\, T \,]\!], t) = \sigma. \qquad (\ddagger)$$

We are ready to prove Equation (5), yielding the soundness of the protocol for the PLMN. It is easy to prove that

$$(\nu l)(l :: \mathbf{login}(\mathtt{M}).\mathbf{out}(\text{'send'}, l', \text{'HI'})@l \parallel l' :: \mathbf{login}(\mathtt{M}) \parallel SYS)$$

$$\begin{aligned}
\simeq_d\ & (\nu l, \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\ l :: \langle \text{'send'}, l', \text{'HI'} \rangle \ \parallel\ l' :: \mathbf{nil} \\
& \quad \parallel\ \mathtt{M} :: *ENTER \mid *HNDVR \ \parallel\ \mathtt{BSlist} :: \langle \mathtt{B}_1 \rangle \mid \langle \mathtt{B}_2 \rangle \\
& \quad \parallel\ \mathtt{Table} :: \langle l, \mathtt{B}_i \rangle \mid \langle l', \mathtt{B}_j \rangle \ \parallel\ \{l \leftrightarrow \mathtt{B}_i\} \ \parallel\ \{l' \leftrightarrow \mathtt{B}_j\} \\
& \quad \parallel\ \mathtt{B}_1 :: *FWD_1 \mid *CLT_1 \ \parallel\ \mathtt{B}_2 :: *FWD_2 \mid *CLT_2\ )
\end{aligned}$$

$$\begin{aligned}
\simeq_d\ & (\nu l, \mathtt{Table}, \mathtt{BSlist}, \mathtt{B}_1, \mathtt{B}_2)(\ l :: \mathbf{nil} \ \parallel\ l' :: \mathbf{nil} \\
& \quad \parallel\ \mathtt{M} :: *ENTER \mid *HNDVR \ \parallel\ \mathtt{BSlist} :: \langle \mathtt{B}_1 \rangle \mid \langle \mathtt{B}_2 \rangle \\
& \quad \parallel\ \mathtt{Table} :: \langle l, \mathtt{B}_i \rangle \mid \langle l', \mathtt{B}_j \rangle \ \parallel\ \{l \leftrightarrow \mathtt{B}_i\} \ \parallel\ \{l' \leftrightarrow \mathtt{B}_j\} \\
& \quad \parallel\ \mathtt{B}_1 :: *FWD_1 \mid *CLT_1 \ \parallel\ \mathtt{B}_2 :: *FWD_2 \mid *CLT_2 \\
& \quad \parallel\ \mathtt{B}_i :: \mathbf{in}(l', !B)@\mathtt{Table}.\mathbf{out}(l', \text{'HI'})@B\ ) \\
\triangleq\ & K
\end{aligned}$$

The first equality can be inferred using laws ($\star$) and ($\ddagger$), Proposition 1.2, laws ($\star$) and ($\star\star$), and Proposition 1.1; the second equality can be inferred using law ($\ddagger$) and Proposition 1.3. Now we cannot proceed equationally: indeed, there are two parallel components that may want to retrieve the tuple $\langle l', \mathtt{B}_j \rangle$ at `Table`, i.e. the process $\mathbf{in}(l', !B)@\mathtt{Table}.\mathbf{out}(l', \text{'HI'})@B$ running at $\mathtt{B}_i$ and the process $HNDVR$ running at `M`. This fact makes Proposition 1.3 not applicable here.

To overcome this problem, we observe that there are only three possible evolutions for K: make a handover for l, make a handover for l', or complete the delivering of the message that l sent to l'. The first evolution is compatible with the latter two ones that, in turn, are mutually exclusive. Thus, let $\mathcal{H}$ be the set of pairs $(N, (\nu l)(l' :: \langle \text{'HI'} \rangle \parallel SYS))$, where N is any reduct of K obtained by giving the precedence to the handover of l' w.r.t. the message delivering. Symmetrically, let $\mathcal{D}$ be the set of pairs $(N, (\nu l)(l' :: \langle \text{'HI'} \rangle \parallel SYS))$, where N is any reduct of K obtained by giving the precedence to the message delivering w.r.t. the handover of l'. Now, it can be easily proved that

$$\{\, (K, (\nu l)(l' :: \langle \text{'HI'} \rangle \parallel SYS)) \,\} \cup \mathcal{H} \cup \mathcal{D}$$

is a bisimulation. By the fact that $\approx_d \subset \simeq_d$ and by transitivity of $\simeq_d$, this suffices to prove Equation (5).

A Theory of System Behaviour in the Presence of Failures

Adrian Francalanza[1] and Matthew Hennessy[1]

University of Sussex, Falmer Brighton BN1 9RH, England,
{adrianf,matthewh}@sussex.ac.uk

Abstract. We develop a behavioural theory of distributed systems in the presence of failures. The framework we use is that of Dπ, a language in which located processes, or agents, may migrate between dynamically created locations. These processes run on a distributed network, in which individual nodes may fail, or the links between them may be broken. The language is extended by a new construct for detecting, and reacting to these failures.
We define a bisimulation equivalence between these systems, based on labelled actions which record, in addition to the effect actions have on the processes, the actual state of the underlying network and the view of this state known to observers. We prove that the equivalence is *fully abstract*, in the sense that two systems will be differentiated if and only if, in some sense, there is a computational context, consisting of a network and an observer, which can see the difference.

1 Introduction

It is generally accepted that location transparency is not attainable over *wide-area networks*, [5], that is large computational infrastructures which may even span the globe. Because of this various *location-aware* calculi and programming languages have arisen in the literature; not only do these emphasise the *distributed* nature of systems but they also assume that the various system components, processes or agents, are aware of their location in the network, and perhaps also aware of some aspect of the underlying network topology. So computations take place at distinct locations, physical or virtual, and processes may migrate between the locations of which they are aware, to participate in such computations.

It is also argued in [5] that *failures*, and the ability to react to them, are also an inevitable facet of these infrastructures, which must be taken into account when designing languages for location-aware computation. The purpose of this paper is to

- invent a simple framework, a distributed process calculus, for describing computations over a distributed network in which individual nodes and links between the nodes are subject to failures
- use this framework to develop a behavioural theory of distributed systems in which these failures are taken into account.

The framework is an extension of the distributed calculus Dπ [10], in which system configurations now take the form

$$\Pi \triangleright N$$

where Π is a representation of the current state of the network, and N describes the current state of the (software) system executing in a distributed manner over the network. Here Π will record the set of nodes in the network, their *status*, that is whether they are *alive* or *dead*, the set of (symmetric) links between these nodes, and whether any of these links are *broken*. On the other hand N will be more or less a standard system description from Dπ, consisting of a collection of communicating *located* processes, which also have the ability to create new locations (and their links in the network), and to migrate between them. We will also augment the language with a construct for reacting to network failures. We believe that this results in a succinct but expressive framework, in which many of the phenomena associated with network failures can be examined; the details may be found in Section 2.

The behavioural theory is takes the form of *(weak) bisimulation equivalence*, [12] based on labelled actions of the form

$$\Pi \triangleright N \xrightarrow{\mu} \Pi' \triangleright N' \tag{1}$$

where the label μ represents the manner in which an observer, also running on the network Π, can interact with the system N. This interaction may change the state of the system, to N', in the usual manner, but it may also affect the nature of the underlying network. For example an observer may extend the network by creating new locations; but we also allow the observer to kill sites, or break links between sites, thereby capturing at least some of the reaction of N to dynamic failures.

However the definition of these actions needs to be relatively sophisticated. Intuitively the action (1) above is meant to simulate the interaction between an observer and the system. But, although the system and the observer may initially share the same view of the underlying network, interactions quickly give rise to situations in which these views *diverge*. In general observers may not be aware of all the nodes, or links, in a network; they think a particular node is inaccessible because they are not aware of links to it; and they may not be aware of the *status* of such an inaccessible node, precisely because it is inaccessible. So in (1) above the network representation Π, needs to record the actual state of the underlying network, and the *observers view* of it.

In Section 3 we present an initial definition of these actions, based on the general approach of [9]. The resulting bisimulation equivalence can be used to demonstrate equivalencies between systems, but we show, by a series of examples, that in general it is too discriminating. In the next section we revise the definition of these actions, essentially by abstracting from internal information present in the action labels, and show that the resulting equivalence is *fully abstract* with respect to an intuitive form of *contextual equivalence*; that is two systems will be differentiated by the bisimulation equivalence if and only if, in some sense, there is a computational context, consisting of a network and an observer, which can see the difference.

2 The language

We assume a set of *variables* Vars, ranged over by $x, y, z, \ldots$ and a separate set of *names*, Names, ranged over by $n, m, \ldots$, which is divided into locations, Locs, ranged

$$\mathbf{T}, \mathbf{U} ::= \mathbf{ch} \mid \langle \mathbf{A}, \mathbf{L} \rangle \qquad \mathbf{A}, \mathbf{B} ::= \mathbf{a} \mid \mathbf{d} \qquad \mathbf{L}, \mathbf{K} ::= \{u_1, \ldots, u_n\}$$

$$P, Q ::= u!\langle V \rangle.P \mid u?(X).P \mid *u?(X).P \mid \mathbf{0} \mid P|Q \mid (u_1 = u_2).P\lceil Q \rceil \mid (\nu n : \mathbf{T})P \mid \mathsf{mv}\ u(P)\lceil Q \rceil \mid \mathsf{break}\ l \mid \mathsf{kill}$$

$$N, M ::= l[\![P]\!] \mid (\nu n : \mathbf{T})N \mid N|M$$

Fig. 1. Syntax of the DπF

over by $l, k, \ldots$ and channels, Chans, ranged over by $a, b, c, \ldots$. Finally we use $u, v, \ldots$ to range over the set of *identifiers*, consisting of either variables and names.

The syntax of DπF is given in Figure 1, where the main syntactic category is that of *systems*, ranged over by M, N; these are essentially a collection of *located processes*, or *agents* $l[\![P]\!]$, but there may also be occurrences of typed *scoped names*, $(\nu n : \mathbf{T})N$. Although we could employ the full power of the type system for Dπ [8], for simplicity we use a very simple notion of type, and adapt it the purpose at hand. Thus if n is used as a channel in N, then $\mathbf{T}$ is simply **ch**; however if it is a location then $\mathbf{T} = \langle \mathbf{A}, \mathbf{L} \rangle$ records it's *status* $\mathbf{A}$, whether it is alive **a** or dead **d**, and the set of locations $\mathbf{L}$ to which it is linked, $\{l_1, \ldots, l_n\}$.

The syntax for agents, P, Q, is an extension of that in Dπ. There is input and output on channels; here V is a tuple of identifiers, and X a tuple of variables, to be interpreted as a pattern. We also have the standard forms of parallel, replicated input, local declarations, and a test for equality between identifiers. The migration construct of Dπ is replaced with *conditional migration* in the spirit of an exception throwing remote communication construct over the TCP level of abstraction[13, 1]. Intuitively $l[\![\mathsf{mv}\ k(P)\lceil Q \rceil]\!]$ attempts to spawn the code P at the location k; however the state of the network might preclude access of location k from l, in which case the residual "exception code" Q is executed at the original site l. Finally we have two new constructs to simulate failures; $l[\![\mathsf{kill}]\!]$ kills the location l, while $k[\![\mathsf{break}\ l]\!]$ results in the link between l and k being broken, if it exists. We are not really interested in programming with these last two operators; but when we come to consider *contextual behaviour* their presence will mean that the behaviour will take account the effects of dynamic failures.

In this extended abstract we will assume the standard notions of *free* and *bound* occurrences of both names and variables, and the associated concepts of α-conversion and *substitution*. Furthermore we will assume that all system terms have no free occurrences of variables.

Reduction semantics: This takes the form of a binary relation

$$\Pi \triangleright N \longrightarrow \Pi' \triangleright N' \tag{2}$$

where Π and Π' are representations of the state of the network. Intuitively this must record the set of locations in existence, whether they are alive or dead, and the exis-

tence of any links between them. We choose what may seem like an overly complicated representation for this information; but our approach will prove to be very convenient when we later give a labelled transition system for the language.

Definition 1 (Link sets). *A link set consists of a finite collection of pairs of locations, $\{l_1 \leftrightarrow k_1, \ldots, l_n \leftrightarrow k_n\}$; note that there is no difference between the links $l \leftrightarrow k$ and $k \leftrightarrow l$.*
Let G be a linkset.

- *We use* $\mathbf{loc}(G)$ *to denote the set of locations occurring in G.*
- $\mathbf{refLoc}(G)$ *is the set of locations l such that* $l \leftrightarrow l$ *is in G*
- *G is said to be reflexive if* $\mathbf{loc}(G) = \mathbf{refLoc}(G)$
- *finally G is semi-reflexive if* $l \leftrightarrow k \in G$ *implies* $l \in \mathbf{refLoc}(G)$ *or* $k \in \mathbf{refLoc}(G)$

We use link sets to represent the current state of a network. It will be convenient to split the representation into two such sets, one between the nodes which are alive, and the other containing any link involving a dead node.

Definition 2 (Simple network representations). *A simple network representation consists of a triple $\langle \mathcal{N}; \mathcal{S}; \mathcal{D} \rangle$ where*

$\mathcal{N}$ *is a set of names denoting all the free names in the system. For bookkeeping purposes this includes both channel and node names; so* $\mathcal{N} = \mathbf{chan}(\mathcal{N}) \cup \mathbf{loc}(\mathcal{N})$.

$\mathcal{S}$**:** *is* reflexive linkset *denoting the* live *nodes of the network* $\mathbf{refLoc}(\mathcal{S})$*, and the links between them.*

$\mathcal{D}$ *is a* semi-reflexive linkset *that denotes the nodes that are* dead *in the network together with the links connected to these dead nodes. The dead nodes may be taken to be* $\mathbf{refLoc}(\mathcal{D})$.

We will assume that all simple network representations satisfy constraints which assure that the three components are mutually consistent.

- $\mathbf{loc}(\mathcal{S}, \mathcal{D}) \subseteq \mathbf{loc}(\mathcal{N}) \times \mathbf{loc}(\mathcal{N})$
- $\mathbf{refLoc}(\mathcal{S}) \cap \mathbf{refLoc}(\mathcal{D}) = \emptyset$ *(a node is either dead or alive)*
- $\mathbf{refLoc}(\mathcal{S}) \cup \mathbf{refLoc}(\mathcal{D}) = \mathbf{loc}(\mathcal{N})$ *(the state of every node is known)*

So we may take Π and Π' in (2) above to be simple network representations. Formally we call pairs $\Pi \triangleright N$ *configurations*, ranged over by the metavariables C, D, E, whenever every free name in N occurs in the name component of Π, and we define reductions to take place between such configurations. Since not every node is interconnected, the reduction semantics is based on the notion of a *livepath* between nodes, which is a chain of links where every intermediate node is alive. We thus say that a node k is *accessible* from node l if there exists a livepath between the two nodes.

For convenience the rules governing these reductions are given in the three separate figures. These rely on certain notation for checking the state of nodes and links in a network, and of updating the network; in this extended abstract this notation is only explained informally, due to lack of space, but the intentions should be straightforward.

The first set of rules, in Figure 2, give the standard rules for (local) communication, and the management of replication, matching and parallelism, and are derived from the

Assuming $\Pi \vdash l$: **alive**

(r-comm)

$$\Pi \triangleright l[\![a!\langle V\rangle.P]\!] \mid l[\![a?(X).Q]\!] \longrightarrow \Pi \triangleright l[\![P]\!] \mid l[\![Q\{^{V}/_{X}\}]\!]$$

(r-rep)

$$\Pi \triangleright l[\![*a?(X).P]\!] \longrightarrow \Pi \triangleright l[\![a?(X).(P \mid *a?(Y).P\{^{Y}/_{X}\})]\!]$$

(r-eq)

$$\Pi \triangleright l[\![(n = n).P\lceil Q\rceil]\!] \longrightarrow \Pi \triangleright l[\![P]\!]$$

(r-neq)

$$\Pi \triangleright l[\![(n = m).P\lceil Q\rceil]\!] \longrightarrow \Pi \triangleright l[\![Q]\!] \quad n \neq m$$

(r-fork)

$$\Pi \triangleright l[\![P|Q]\!] \longrightarrow \Pi \triangleright l[\![P]\!] \mid l[\![Q]\!]$$

Fig. 2. Local Reduction Rules(1)

Assuming $\Pi \vdash l$: **alive**

(r-move)

$$\Pi \triangleright l[\![\mathsf{mv}\ k(P)\lceil Q\rceil]\!] \longrightarrow \Pi \triangleright k[\![P]\!] \quad \Pi \vdash k : \textbf{alive},\ l \leftrightarrow k$$

(r-nmove)

$$\Pi \triangleright l[\![\mathsf{mv}\ k(P)\lceil Q\rceil]\!] \longrightarrow \Pi \triangleright l[\![Q]\!] \quad \Pi \nvdash k : \textbf{alive},\ l \leftrightarrow k$$

(r-new)

$$\Pi \triangleright l[\![(\nu n : \mathbf{U})P]\!] \longrightarrow \Pi \triangleright (\nu n : \mathbf{T})l[\![P]\!] \quad \mathbf{T} = \mathsf{inst}(\mathbf{U}, l, \Pi)$$

(r-kill)

$$\Pi \triangleright l[\![\mathsf{kill}]\!] \longrightarrow (\Pi - l) \triangleright l[\![\mathbf{0}]\!]$$

(r-brk)

$$\Pi \triangleright l[\![\mathsf{break}\ k]\!] \longrightarrow (\Pi - l \leftrightarrow k) \triangleright l[\![\mathbf{0}]\!] \quad \Pi \vdash l \leftrightarrow k$$

Fig. 3. Network Reduction Rules (2) for DπF

corresponding rules for Dπ in [10]. But note that they are all depend on the requirement that l, the location of the activity, is currently alive; this is the intent of the predicate $\Pi \vdash l$: **alive**.

The second set, in Figure 3, is more interesting. Again all the reductions rely on the assumption that the site of focus l is alive. The first rule, (r-move), says that $\mathsf{mv}\ k(P)\lceil Q\rceil$, running at location l can spawn agent P, at k, *provided k is accessible from l* in the current network Π; This is the import of the side conditions $\Pi \vdash k$: **alive**, $l \leftrightarrow k$. Similarly if this condition is not true, by (r-nmove) the residual Q is launched locally at l. The rules (r-kill), (r-brk) make the obvious changes to the current network. $(\Pi - l)$ means

changing l to be a dead site in Π if it is alive, while $\Pi - l \leftrightarrow k$ means removing the link between l and k if it exists; for lack of space we omit the formal definitions. Finally (r-new) regulates the generation of new names; if $\mathbf{U}$ is simply $\mathbf{ch}$ then $\mathsf{inst}(\mathbf{U}, l, \Pi)$ also evaluates to $\mathbf{ch}$; if $\mathbf{U}$ is a location type $\langle \mathbf{A}, \mathbf{L} \rangle$, then $\mathsf{inst}(\mathbf{U}, l, \Pi)$ returns the location type $\mathbf{T} = \langle \mathbf{A}, \mathbf{K} \rangle$, where $\mathbf{K}$ is the set of locations in $\mathbf{L}$ *accessible* to l in the current network Π.

(r-str)
$$\frac{\Pi \triangleright N' \equiv \Pi \triangleright N \quad \Pi \triangleright N \longrightarrow \Pi' \triangleright M \quad \Pi' \triangleright M \equiv \Pi' \triangleright M'}{\Pi \triangleright N' \longrightarrow \Pi' \triangleright M'}$$

(r-ctxt-rest)
$$\frac{\Pi + n : \mathbf{T} \triangleright N \longrightarrow \Pi' + n : \mathbf{U} \triangleright M}{\Pi \triangleright (\nu n : \mathbf{T})N \longrightarrow \Pi' \triangleright (\nu n : \mathbf{U})M}$$

(r-ctxt-par)
$$\frac{\Pi \triangleright N \longrightarrow \Pi' \triangleright N'}{\begin{array}{l}\Pi \triangleright N|M \longrightarrow \Pi' \triangleright N'|M \\ \Pi \triangleright M|N \longrightarrow \Pi' \triangleright M|N'\end{array}}\ \Pi \vdash M$$

Fig. 4. Contextual Reduction Rules(3)

Finally in Figure 4 we have an adaptation of the standard *contextual* rules, which allow the basic reductions to occur in *evaluation contexts*. The rule (r-str) allows reductions up to a structural equivalence, in the standard manner. This equivalence is the least one generated in the usual way from the identities in Figure 5 defined over systems, that extends naturally to configurations with the same network Π. The only non-trivial identity in Figure 5 is (s-flip-2), in which the types of the successively scoped locations need to be changed if they denote a link between them, thus avoiding unwanted name capture. The rules (r-ctxt-par) and (r-ctxt-rest) allow reductions to occur under contexts; note that the latter is somewhat non-standard, but as reductions can change the underlying network, it may be that the connectivity of the scoped name n is affected by the reduction, thereby changing $\mathbf{T}$ to $\mathbf{U}$.

(s-comm)	$N\|M \equiv M\|N$	
(s-assoc)	$(N\|M)\|M' \equiv N\|(M\|M')$	
(s-unit)	$N\|l[\![\mathbf{0}]\!] \equiv N$	
(s-extr)	$(\nu n : \mathbf{T})(N\|M) \equiv N\|(\nu n : \mathbf{T})M$	$n \notin \mathbf{fn}(N)$
(s-flip-1)	$(\nu n : \mathbf{T})(\nu m : \mathbf{U})N \equiv (\nu m : \mathbf{U})(\nu n : \mathbf{T})N$	$n \notin \mathbf{U}$
(s-flip-2)	$(\nu l : \langle \mathbf{A}, \mathbf{L} \rangle)(\nu k : \langle \mathbf{B}, \mathbf{K} \cup \{l\} \rangle)N \equiv (\nu k : \langle \mathbf{B}, \mathbf{K} \rangle)(\nu l : \langle \mathbf{A}, \mathbf{L} \cup \{k\} \rangle)N$	

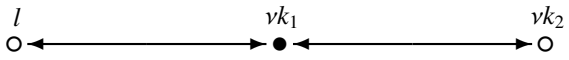
Fig. 5. Structural Rules for Dπ_f

This completes our exposition of the reduction semantics. But at this point we should point out that in a configuration such as $\Pi \triangleright N$, contrary to what we have implied up to now, Π does not give a completely true representation of the network on which the code in N is running.

Example 1. Let Π represent the network $\langle\{l, a\}; \{l \leftrightarrow l\}; \emptyset\rangle$ consisting of a channel a and a live node l and M the system

$$(\nu k_2 : \langle \mathbf{a}, \emptyset\rangle)(\nu k_1 : \langle \mathbf{d}, \{l, k_2\}\rangle) l[\![a!\langle k_2\rangle.P]\!]$$

Here M generates two new locations $k_1,\ k_2$, where k_1 is dead and linked to the existing node l and k_2 is alive linked to k_1. Although Π only contains one node l, effectively the located process $l[\![a!\langle k_2\rangle.P]\!]$ is running on a network of *three nodes*, two of which, $k_1,\ k_2$ are scoped, that is not available to other systems. We can informally represent this network by



where the nodes ∘ and • denote live and dead nodes respectively. At this point we note that the same network could be denoted by the system M'

$$(\nu k_1 : \langle \mathbf{d}, \{l\}\rangle)(\nu k_2 : \langle \mathbf{a}, \{k_1\}\rangle) l[\![a!\langle k_2\rangle.P]\!]$$

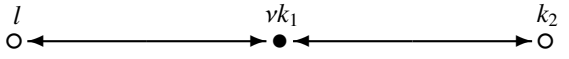
Note also that the two systems are structurally equivalent, $M \equiv M'$, through (s-flip-2). As a notational abbreviation, in all future example we will omit the status annotation, **a**, in location declarations; so for example system M' would be given as

$$(\nu k_1 : \langle \mathbf{d}, \{l\}\rangle)(\nu k_2 : \{k_1\}) l[\![a!\langle k_2\rangle.P]\!]$$

3 A Labelled transition system

In this section we give a labelled transition system for the language, in which the labelled actions are intended to mimic the possible interactions between a system and an observer; it is natural to assume that both share the same underlying network. However this first example demontrates that our representation of this joint network is no longer sufficient, if we want to faithfully record the effect interactions have on systems.

Example 2. Let Π and M be defined as in the last example. An observer N at site l, such as $l[\![a?(x).P(x)]\!]$, can gain knowledge of the new location k_2, thereby evolving to $l[\![P(k_2)]\!]$. But even though it is in possession of the name k_2, it's knowledge of the state of the underlying network is no longer represented by Π, and there is now a mismatch between the observes view of the network, and the systems view. The system view is now $\Pi' = \langle\{a, l, k_2\}; \{l \leftrightarrow l,\ k_2 \leftrightarrow k_2\}; \emptyset\rangle$, or Π augmented by the scope extrusion of the *live* node k_2 linked to a private (dead) node k_1, which in turn is linked to the node l; informally it may be depicted as



In this diagram the open nodes l, k_2 are alive and accessible for the system to use; moreover they are linked via a private node k_1. But the observer's view is quite different. The node l is accessible to the observer, since it has code running there; on the other hand, even though the observer knows about k_2 at l in $P(k_2)$, it does not have enough information to determine a livepath to access k_2 from l. As a result, it has no means how to determine k_2's state in terms of its status and connections. This means that the representation of the observers view, requires a new kind of annotation, for nodes such as k_2 which are known, but not accessible:

This example demonstrates is that in order to give an lts semantics we need to extend our representations of networks.

Definition 3 (Network representations). *A* network representation *is a 4-tuple* $\langle \mathcal{N}; \mathcal{S}; \overline{\mathcal{S}}; \mathcal{D} \rangle$*, where* $\mathcal{N}$, $\mathcal{S}$ *and* $\mathcal{D}$ *are as in Definition 2, and* $\overline{\mathcal{S}}$ *is another* reflexive linkset *that denotes the* live *nodes that are* not accessible *to the observer together with the links between these inaccessible nodes.*

Note that simple network configurations can be viewed as a degenerate form of network representations, where the $\overline{\mathcal{S}}$ component is empty. Also the reduction semantics of the previous section can be viewed as a relation over configurations; it is simply a matter of extending the various side conditions on the rules from simple network representations to full representations. The operation $\uparrow(\Pi)$ for $\Pi = \langle \mathcal{N}; \mathcal{S} \rangle \overline{\mathcal{S}} \mathcal{D}$ returns a network with no hidden live nodes, $\langle \mathcal{N}; \mathcal{S} \cup \overline{\mathcal{S}} \rangle \emptyset \mathcal{D}$.

With this extended notion we can now represent the observers view of the network above; $\mathcal{N} = \{a, l, k_2\}$, $\mathcal{S} = \{l \leftrightarrow l\}$, $\overline{\mathcal{S}} = \{k_2 \leftrightarrow k_2\}$ and $\mathcal{D} = \emptyset$. So in the sequel we will use *configurations* of the form $\Pi \triangleright N$, where Π is a network representation, and N satisfies the obvious consistency constraints with respect to it.

We now define a labelled transition system for DπF, which consists of a collection of actions over configurations, $C \xrightarrow{\mu} D$, defined by the transition rules in Figures 6, 7 and 8, where μ can take one of the forms: -

- τ (internal action)
- $(\tilde{n} : \tilde{\mathbf{T}})l : a?(V)$ (bound input)
- $(\tilde{n} : \tilde{\mathbf{T}})l : a!\langle V \rangle$ (bound output)
- $\mathsf{kill}(l)$ (external location killing)
- $l \nleftrightarrow k$ (external breaking of the link between locations l and k by an observer)

The definitions of these actions are modelled on those in [9, 8], and in this extended abstract we refrain form commenting on them. Many are inherited directly from the reductions semantics, although the side-conditions now apply to network representations in general. We only highlight the fact that the transition rules introducing external actions such as (l-out), (l-in) for external output and input, and (l-halt), (l-disc) for external killing of nodes and breaking of links, are subject to judgements of the form $\Pi \vdash_{\mathsf{obs}} l$: **alive** requiring that l is alive and *accessible* by the observer.

Assuming $\Pi \vdash l : \mathbf{alive}$

(l-out)

$$\frac{}{\Pi \triangleright l[\![a!\langle V\rangle.P]\!] \xrightarrow{l:a!\langle V\rangle} \Pi \triangleright l[\![P]\!]}\ \Pi \vdash_{\mathsf{obs}} l : \mathbf{alive}$$

(l-in)

$$\frac{}{\Pi \triangleright l[\![a?(X).P]\!] \xrightarrow{l:a?(V)} \Pi \triangleright l[\![P\{^{V}/_{X}\}]\!]}\ \Pi \vdash_{\mathsf{obs}} l : \mathbf{alive}, V$$

(l-in-rep)

$$\frac{}{\Pi \triangleright l[\![*a?(X).P]\!] \xrightarrow{\tau} \Pi \triangleright l[\![a?(X).(P \mid *a?(Y).P\{^{Y}/_{X}\})]\!]}$$

(l-eq)

$$\frac{}{\Pi \triangleright l[\![(n = n).P\lceil Q\rceil]\!] \xrightarrow{\tau} \Pi \triangleright l[\![P]\!]}$$

(l-neq)

$$\frac{}{\Pi \triangleright l[\![(n = m).P\lceil Q\rceil]\!] \xrightarrow{\tau} \Pi \triangleright l[\![Q]\!]}\ n \neq m$$

(l-fork)

$$\frac{}{\Pi \triangleright l[\![P|Q]\!] \xrightarrow{\tau} \Pi \triangleright l[\![P]\!] \mid l[\![Q]\!]}$$

Fig. 6. Local Operational Rules(1) for Dπ_f

Assuming $\Pi \vdash l : \mathbf{alive}$

(l-kill)

$$\frac{}{\Pi \triangleright l[\![\mathsf{kill}]\!] \xrightarrow{\tau} (\Pi - l) \triangleright l[\![\mathbf{0}]\!]}$$

(l-brk)

$$\frac{}{\Pi \triangleright l[\![\mathsf{break}\ k]\!] \xrightarrow{\tau} \Pi - (l \leftrightarrow k) \triangleright l[\![\mathbf{0}]\!]}\ \Pi \vdash l \leftrightarrow k$$

(l-halt)

$$\frac{}{\Pi \triangleright N \xrightarrow{\mathsf{kill}(l)} (\Pi - l) \triangleright N}\ \Pi \vdash_{\mathsf{obs}} l : \mathbf{alive}$$

(l-disc)

$$\frac{}{\Pi \triangleright N \xrightarrow{l \leftrightarrow k} \Pi - (l \leftrightarrow k) \triangleright N}\ \Pi \vdash_{\mathsf{obs}} l \leftrightarrow k$$

(l-new)

$$\frac{}{\Pi \triangleright l[\![(\nu n : \mathbf{U})P]\!] \xrightarrow{\tau} \Pi \triangleright (\nu n : \mathbf{T})l[\![P]\!]}\ \mathbf{T} = \mathsf{inst}(\mathbf{U}, l, \Pi)$$

(l-move)

$$\frac{}{\Pi \triangleright l[\![\mathsf{mv}\ k(P)\lceil Q\rceil]\!] \xrightarrow{\tau} \Pi \triangleright k[\![P]\!]}\ \Pi \vdash k : \mathbf{alive}, l \leftrightarrow k$$

(l-nmove)

$$\frac{}{\Pi \triangleright l[\![\mathsf{mv}\ k(P)\lceil Q\rceil]\!] \xrightarrow{\tau} \Pi \triangleright l[\![Q]\!]}\ \Pi \not\vdash k : \mathbf{alive}, l \leftrightarrow k$$

Fig. 7. Network Operational Rules(2) for DπF

With these actions we can now define in the standard manner a bisimulation equivalence between configurations, which can be used as the basis for contextual reasoning.

(l-open)

$$\frac{\Pi + n : \mathbf{T} \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}})l:a!\langle V\rangle} \Pi' \triangleright N'}{\Pi \triangleright (\nu n : \mathbf{T})N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}},n:\mathbf{T})l:a!\langle V\rangle} \Pi' \triangleright N'}\ l, a \neq n \in V$$

(l-weak)

$$\frac{\Pi + n : \mathbf{T} \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}})l:a?(V)} \Pi' \triangleright N'}{\Pi \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}},n:\mathbf{T})l:a?(V)} \Pi' \triangleright N'}\ l, a \neq n \in V,\ \Pi \vdash_{\mathsf{obs}} \tilde{n} : \tilde{\mathbf{T}}, n : \mathbf{T}$$

(l-rest)

$$\frac{\Pi + n : \mathbf{T} \triangleright N \xrightarrow{\mu} \Pi' + n : \mathbf{U} \triangleright N'}{\Pi \triangleright (\nu n : \mathbf{T})N \xrightarrow{\mu} \Pi' \triangleright (\nu n : \mathbf{U})N'}\ n \notin \mathbf{fn}(\mu)$$

(l-rest-typ)

$$\frac{\Pi + k : \mathbf{T} \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}})l:a!\langle V\rangle} (\Pi + \tilde{n} : \tilde{\mathbf{U}}) + k : \mathbf{U} \triangleright N'}{\Pi \triangleright (\nu k : \mathbf{T})N \xrightarrow{(\tilde{n}:\tilde{\mathbf{U}})l:a!\langle V\rangle} \Pi + \tilde{n} : \tilde{\mathbf{U}} \triangleright (\nu k : \mathbf{U})N'}\ l, a \neq k \in \mathbf{n}(\tilde{\mathbf{T}})$$

(l-par-ctxt)

$$\frac{\Pi \triangleright N \xrightarrow{\mu} \Pi' \triangleright N'}{\Pi \triangleright N|M \xrightarrow{\mu} \Pi' \triangleright N'|M}\ \Pi \vdash M$$
$$\Pi \triangleright M|N \xrightarrow{\mu} \Pi' \triangleright M|N'$$

(l-par-comm)

$$\frac{\uparrow(\Pi) \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}})l:a!\langle V\rangle} \Pi' \triangleright N' \qquad \uparrow(\Pi) \triangleright M \xrightarrow{(\tilde{n}:\tilde{\mathbf{U}})l:a?(V)} \Pi'' \triangleright M'}{\begin{array}{l}\Pi \triangleright N|M \xrightarrow{\tau} \Pi \triangleright (\nu\tilde{n} : \tilde{\mathbf{T}})(N'|M') \\ \Pi \triangleright M|N \xrightarrow{\tau} \Pi \triangleright (\nu\tilde{n} : \tilde{\mathbf{T}})(M'|N')\end{array}}$$

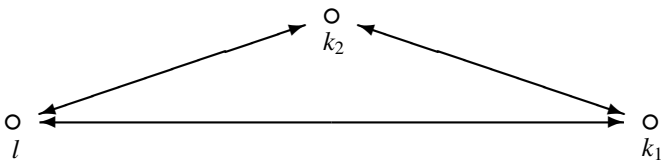
Fig. 8. Contextual Operational Rules(3) for DπF

Let us write

$$\Pi \models M \approx_{int} N$$

to mean that there is a (weak) bisimulation between the configurations $\Pi \triangleright M$ and $\Pi \triangleright N$

Example 3. Consider the (simple) network:



formally represented as $\Pi = \langle \mathcal{N}; \mathcal{S}; \overline{\mathcal{S}}; \mathcal{D}\rangle$, where $\mathcal{N} = \{l, k_1, k_2, serv, ans\}$, $\mathcal{S} = \{l \leftrightarrow l,\ k_1 \leftrightarrow k_1,\ k_2 \leftrightarrow k_2,\ l \leftrightarrow k_1,\ l \leftrightarrow k_2,\ k_1 \leftrightarrow k_2\}$ and $\overline{\mathcal{S}} = \mathcal{D} = \emptyset$. Running on Π

we consider two systems that implement two variants of a simple remote client server accepting a request on *serv* at l, querying remote *data* at k_1 and returning an answer on *ans* at l. We use $\mathsf{mv}\ l(P)\ \mathsf{mv}\ l, k(P)$ as shorthand for $\mathsf{mv}\ l(P)\lceil\mathbf{0}\rceil$ and $\mathsf{mv}\ l(\mathsf{mv}\ k(P))$ respectively.

$$
\begin{aligned}
C_{rcs} \Leftarrow \Pi \triangleright\ & l[\![serv!\langle a\rangle]\!] \\
& \mid (\nu data)(\ l[\![serv?(x).\mathsf{mv}\ k_1(data!\langle x\rangle)]\!] \\
& \quad \mid k_1[\![data?(x).\mathsf{mv}\ l(ans!\langle f(x)\rangle)]\!]\ ) \\
D_{rcs} \Leftarrow \Pi \triangleright\ & l[\![serv!\langle a\rangle]\!] \\
& \mid (\nu data)(\ l[\![serv?(x).\mathsf{mv}\ k_1(data!\langle x\rangle)\lceil\mathsf{mv}\ k_2, k_1(data!\langle x\rangle)\rceil]\!] \\
& \quad \mid k_1[\![data?(x).\mathsf{mv}\ l(ans!\langle f(x)\rangle)\lceil\mathsf{mv}\ k_2, l(ans!\langle f(x)\rangle)\rceil]\!])
\end{aligned}
$$

The main difference between the two implementations is that D_{rcs} tries to access the remote resource *data* using two paths while C_{rcs} only tries the direct route from l to k_1. It is straightforward to argue that these systems are not bisimilar, when running relative to Π; formally

$$\Pi \models C_{rcs} \not\approx_{int} D_{rcs}$$

To see this, it is sufficient to examine the behaviour of the two configurations subsequent to an action such as $\xrightarrow{l \leftrightarrow k_1}$. However, in a calculus where link failure are not considered, we would not be able to distinguish between these two implementations.

One can also use the lts to establish positive results. For example, over the same network Π used in Example 3, one can establish

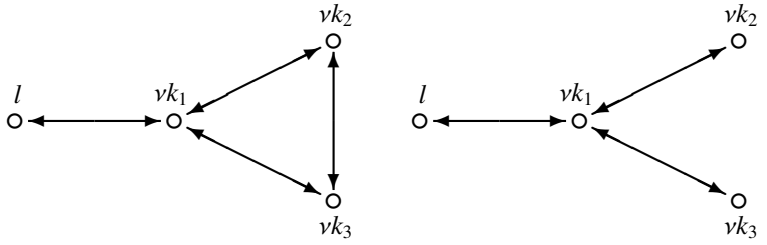
$$\Pi \models k_2[\![\mathsf{mv}\ k_1(\mathsf{mv}\ l(ans!\langle\rangle))]\!] \approx_{int} k_1[\![\mathsf{mv}\ k_2(\mathsf{mv}\ l(ans!\langle\rangle))]\!]$$

However we can argue, at least informally, that this notion of equivalence is too discriminating and the labels too intentional, in that we distinguish between systems running on a network, where the differences in behaviour are difficult to observe. Problems arise when there is an interplay between *hidden* nodes, links and dead nodes.

Example 4. Let Π be the (simple) configuration in which there is only one node l which is alive, and consider the two systems

$$
\begin{aligned}
M &\Leftarrow (\nu k_1 : \{l\})(\nu k_2 : \{k_1\})(\nu k_3 : \{k_1, k_2\})l[\![a!\langle k_2, k_3\rangle.P]\!] \\
N &\Leftarrow (\nu k_1 : \{l\})(\nu k_2 : \{k_1\})(\nu k_3 : \{k_1\})l[\![a!\langle k_2, k_3\rangle.P]\!]
\end{aligned}
$$

Note that when M and N are running on Π, because of the new locations declared, the code $l[\![a!\langle k_2, k_3\rangle.P]\!]$ is effectively running on the following networks respectively:



It turns out that

$$\Pi \models M \not\approx_{int} N$$

because the configurations give rise to *different* output actions, on a at l. The difference lies in the types at which the locations k_2 and k_3 are exported; in $\Pi \triangleright M$ the output label is $\mu_1 = (k_2 : \emptyset,\ k_3 : \{k_2\})l : a!\langle k_2, k_3\rangle$ while with $\Pi \triangleright N$ it is $\mu_2 = (k_2 : \emptyset,\ k_3 : \emptyset)l : a!\langle k_2, k_3\rangle$. We note that the derivation of both these actions require applications of the tricky rule (l-rest-typ); as an illustration we here give the last derivation step for the transition of configuration $\Pi \triangleright M$.

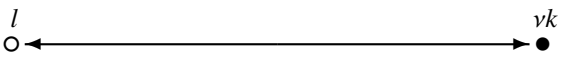
$$\frac{\dfrac{\dots}{(\Pi + k_1 : \{l\}) \triangleright M' \xrightarrow{\mu_3} ((\Pi + k_2 : \emptyset) + k_3 : \{k_2\}) + k_1 : \{l, k_2, k_3\} \triangleright l[\![P]\!]}}{\Pi \triangleright (\nu k_1 : \{l\})M' \xrightarrow{\mu_1} ((\Pi + k_2 : \emptyset) + k_3 : \{k_2\}) \triangleright (\nu k_1 : \{l, k_2, k_3\})l[\![P]\!]}\ \text{(l-rest-typ)}$$

where $M' = (\nu k_2 : \{k_1\})(\nu k_3 : \{k_1, k_2\})l[\![a!\langle k_2, k_3\rangle.P]\!]$ and $\mu_3 = (k_2 : \{l\},\ k_3 : \{l, k_2\})l : a!\langle k_2, k_3\rangle$

However if k_1 does not occur in P, then k_1 can never be scope extruded to the observer and thus k_2 and k_3 will remain inaccessible in both systems. This means that the presence (or absence) of the link $k_2 \leftrightarrow k_3$ can never be checked and thus there should be no observable difference between M and N running on Π.

Example 5. We consider the following three configuration definitions together with the representation of their respective networks

$$E_1 \Leftarrow \langle\{l, a\}; \{l_1 \leftrightarrow l_1\}; \emptyset; \emptyset\rangle \triangleright (\nu k : \langle \mathbf{d}, \{l\}\rangle)l[\![a!\langle k\rangle.P]\!]$$



$$E_2 \Leftarrow \langle\{l, a\}; \{l_1 \leftrightarrow l_1\}; \emptyset; \emptyset\rangle \triangleright (\nu k : \langle \mathbf{d}, \emptyset\rangle)l[\![a!\langle k\rangle.P]\!]$$

$$E_3 = \langle\{l, a\}; \{l_1 \leftrightarrow l_1\}; \emptyset; \emptyset\rangle \triangleright (\nu k : \langle \mathbf{a}, \emptyset\rangle)l[\![a!\langle k\rangle.P]\!]$$

Intuitively no observer can distinguish between these three configurations; even though some observer might obtain the scoped name k by inputting on channel $a@l$, it cannot determine the difference in the state of network. From rule (l-nmove) we conclude that any attempt to move to k from l will fail. However, such a failure does not yield the observer enough information about whether it was caused by a node fault at k, a link fault between l and k or both. As a result, we would like to equate all three configuration. However, our lts specifies that all three configurations perform the output with different scope extrusion labels, namely

$$E_1 \xrightarrow{(k:\langle \mathbf{d}, \{l\}\rangle)l:a!\langle k\rangle} \langle\{l, a\}; \{l_1 \leftrightarrow l_1\}; \emptyset; \{k \leftrightarrow k, k \leftrightarrow l\}\rangle \triangleright l[\![P]\!]$$
$$E_2 \xrightarrow{(k:\langle \mathbf{d}, \emptyset\rangle)l:a!\langle k\rangle} \langle\{l, a\}; \{l_1 \leftrightarrow l_1\}; \emptyset; \{k \leftrightarrow k\}\rangle \triangleright l[\![P]\!]$$
$$E_3 \xrightarrow{(k:\langle \mathbf{a}, \emptyset\rangle)l:a!\langle k\rangle} \langle\{l, a\}; \{l_1 \leftrightarrow l_1\}; \{k \leftrightarrow k\}; \emptyset\rangle \triangleright l[\![P]\!]$$

4 Reduction barbed congruence

The essential problem with the lts of the previous section is that when new locations are created, the associated information, coded in the types at which they are exported, is too detailed. Intuitively a network representation $\Pi = \langle \mathcal{N}; \mathcal{S}; \overline{\mathcal{S}}; \mathcal{D} \rangle$ involves both *externally visible* or *accessible* information, $\mathcal{N}$ and $\mathcal{S}$, and *internal* information, $\overline{\mathcal{S}}$ and $\mathcal{D}$; the latter may only be used by the system itself. The main problem with the current actions is that they carry too much *internal* information. We need a revised form of actions, which carry just the right amount of information.

But before we plunge into our revision, it is best to have yardstick with respect to which we can calibrate the appropriateness of the revised labelled actions, and the resulting bisimulation equivalence. We adapt a well-known formulation of contextual equivalence to DπF, [11, 9], called *reduction barbed congruence*. This relies on the notion of a *barb*, a collection of primitive observations which can be made on systems. Let us write $\Pi \triangleright N \Downarrow_{a@l}$ to mean that an output on channel a at the live location l can be observed. Then we would expect all reasonable behavioural equivalencies to preserve these barbs.

But the key idea in the definition is to use a notion of *contextual* relation over configurations, in which the contexts only have access to the *observable* part of the network.

Definition 4. *A relation $\mathcal{R}$ over configurations is* contextual *if*

- $\Pi \triangleright M \ \mathcal{R} \ \Pi' \triangleright N$ *and* $\Pi,\ \Pi' \vdash_{obs} O$ *implies* $\Pi \triangleright M|O \ \mathcal{R} \ \Pi' \triangleright N|O$ *and* $\Pi \triangleright O|M \ \mathcal{R} \ \Pi' \triangleright O|N$
- $(\Pi + n : \mathbf{T}) \triangleright M \ \mathcal{R} \ (\Pi' + n : \mathbf{U}) \triangleright N$ *implies* $\Pi \triangleright (\nu n : \mathbf{T})M \ \mathcal{R} \ \Pi' \triangleright (\nu n : \mathbf{U})N$
- $\Pi \triangleright M \ \mathcal{R} \ \Pi' \triangleright N$ *and* $\Pi,\ \Pi' \vdash_{obs} \mathbf{T}$ *implies* $(\Pi + n : \mathbf{T}) \triangleright M \ \mathcal{R} \ (\Pi' + n : \mathbf{T}) \triangleright N$, *whenever n is fresh.*

In this first clause the constraints, $\Pi \vdash_{\mathsf{obs}} O$ and $\Pi' \vdash_{\mathsf{obs}} O$ ensure that only observers with access to the *external* components of the networks are used. Note also that the third clause ensures that the behaviour of M and N are also examined when they are placed in *larger* networks.

Definition 5 (Reduction barbed congruence). *Let $\cong$ be the largest relation between configurations which is* contextual*, preserves* barbs *and is* reduction-closed*.*

Note that apriori this definition allows us to compare configurations which have different networks. However it turns out that whenever $\Pi \triangleright M \cong \Pi' \triangleright N$, the external parts of Π and Π' must coincide. In the sequel we will abbreviate $\Pi \triangleright M \cong \Pi \triangleright N$ to $\Pi \models M \cong N$.

We now outline a revision of our labelled actions with the property that the resulting bisimulation equivalence coincides with this yardstick relation $\cong$. The idea is to reuse the same actions but to simply change the types at which bound names appear. Currently these are of the form $\mathbf{T} = \mathbf{ch}$ or $\langle \mathbf{A}, \{k_1, \ldots k_n\} \rangle$, where the latter indicates the liveness of a location and the nodes k_i to which it is linked. We change these types to new types of the form $\mathbf{S}, \mathbf{R} = \{l_1 \leftrightarrow k_1, \ldots, l_i \leftrightarrow k_i\}$ where $\mathbf{S}$ is a semi-reflexive linkset. Intuitively, $\mathbf{S}$ represent the new live nodes, $\mathbf{refLoc}(\mathbf{S})$, which are made accessible to observers by the extrusion of the new location together with links between these nodes and nodes

(l-deriv-1)

$$\frac{\Pi \triangleright N \xrightarrow{\mu} \Pi' \triangleright N'}{\Pi \triangleright N \overset{\mu}{\longmapsto} \Pi' \triangleright N'}\ \mu \in \{\tau, \mathsf{kill}(l), l \leftrightarrow k\}$$

(l-deriv-2)

$$\frac{\Pi \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}})l:a!\langle V\rangle} \Pi' \triangleright N'}{\Pi \triangleright N \overset{(\tilde{n}:\tilde{\mathbf{S}})l:a!\langle V\rangle}{\longmapsto} \Pi' \triangleright N'}\ \tilde{\mathbf{S}} = \mathsf{netType}(n, \tilde{\mathbf{T}}, \Pi)$$

(l-deriv-3)

$$\frac{\Pi \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathbf{T}})l:a?(V)} \Pi' \triangleright N'}{\Pi \triangleright N \overset{(\tilde{n}:\tilde{\mathbf{S}})l:a?(V)}{\longmapsto} \Pi' \triangleright N'}\ \tilde{\mathbf{S}} = \mathsf{netType}(n, \tilde{\mathbf{T}}, \Pi)$$

Fig. 9. The derived lts

which were already observable; alternatively, $\mathbf{S}$ is the information which is added to the external part of the network representation as a result of the action.

The formal definition is given in Figure 9, which is expressed in terms of a function $\mathsf{netType}(n, \mathbf{T}, \Pi)$. If n is a channel ($\mathbf{T} = \mathbf{ch}$) or a dead location ($\mathbf{T} = \langle \mathbf{d}, \mathbf{L}\rangle$), this returns the empty link set $\emptyset$. Otherwise, when it is a live location ($\mathbf{T} = \langle \mathbf{a}, \mathbf{L}\rangle$), it constructs the linkset denoting the nodes and links that are made accessible by the addition of the new location $n : \langle \mathbf{a}, \mathbf{L}\rangle$ to the network Π. In this extended abstract we omit the formal definition of $\mathsf{netType}(n, \mathbf{T}, \Pi)$ due to lack of space.

These revised actions give rise to a new bisimulation equivalence over configurations, $\approx$, and we use

$$\Pi \models M \approx N$$

to mean that the configurations $\Pi \triangleright M$ and $\Pi \triangleright N$ are bisimilar.

Example 6. Here we re-examine the systems in Example 4 and Example 5. We recall that in Example 4 we had the following actions with respect to the original lts: -

$$\Pi \triangleright M \xrightarrow{\mu_1} ((\Pi + k_2 : \emptyset) + k_3 : \{k_2\}) \triangleright (\nu k_1 : \{l, k_2, k_3\})l[\![P]\!]$$

$$\Pi \triangleright N \xrightarrow{\mu_2} ((\Pi + k_2 : \emptyset) + k_3 : \emptyset) \triangleright (\nu k_1 : \{l, k_2, k_3\})l[\![P]\!]$$

But Π contains only one accessible node l; extending it with the new node k_2, linked to nothing does not increase the set of accessible nodes. Further increasing it with a new node k_3, linked to the inaccessible k_2 (in the case of $\Pi \triangleright M$) or completely disconnected (in the case of $\Pi \triangleright N$), also leads to no increase in the accessible nodes. Correspondingly, the calculations of $\mathsf{netType}(k_2, \emptyset, \Pi)$ and $\mathsf{netType}(k_3, \{k_2\}, (\Pi + k_2 : \emptyset))$ both lead to the empty link set. Formally we get the same derived actions

$$\Pi \triangleright M \overset{\alpha}{\longmapsto} ((\Pi + k_2 : \emptyset) + k_3 : \{k_2\}) \triangleright (\nu k_1 : \{l, k_2, k_3\})l[\![P]\!]$$

$$\Pi \triangleright N \overset{\alpha}{\longmapsto} ((\Pi + k_2 : \emptyset) + k_3 : \emptyset) \triangleright (\nu k_1 : \{l, k_2, k_3\})l[\![P]\!]$$

where α is $(k_2 : \emptyset, k_3 : \emptyset)l : a!\langle k_2, k_3\rangle$. Furthermore if P contains no occurrence of k_1, we can go on to show

$$\Pi \models M \approx N$$

On the other hand, if P is $a!\langle k_1\rangle$, the subsequent transitions are:-

$$((\Pi + k_2 : \emptyset) + k_3 : \{k_2\}) \triangleright (\nu k_1 : \{l, k_2, k_3\})l[\![P]\!] \xrightarrow{\beta_1} \ldots$$
$$((\Pi + k_2 : \emptyset) + k_3 : \emptyset) \triangleright (\nu k_1 : \{l, k_2, k_3\})l[\![P]\!] \xrightarrow{\beta_2} \ldots$$

where β_1 is $(k_1 : \mathbf{S})l : a!\langle k_1\rangle$ and β_2 is $(k_1 : \mathbf{R})l : a!\langle k_1\rangle$ where $\mathbf{S}/\mathbf{R} = \{k_2 \leftrightarrow k_3\}$. More specifically, $\mathbf{S}$ and $\mathbf{R}$ hold information directly related to k_1 such as $k_1 \leftrightarrow k_1, k_1 \leftrightarrow l, \ldots$ together with information related to other newly accessible nodes such as $k_2 \leftrightarrow k_2, k_2 \leftrightarrow k_3, \ldots$. The derived action β_1 exports an extra link $k_2 \leftrightarrow k_3$ in $\mathbf{S}$ and based on this discrepancy between β_1 and β_2 we are allowed to discriminate M and N, and thus

$$\Pi \models M \not\approx N$$

Revisiting Example 5, the three different actions of E_1, E_2 and E_3 now converge to the same action $E_i \xrightarrow{\alpha} \ldots \triangleright l[\![P]\!]$ where α is the label $(k : \emptyset)l : a!\langle k\rangle$.

The main result of this paper can now be stated:

Theorem 1. *In DπF, $\Pi \models M \approx N$ if and only if $\Pi \models M \cong N$*

Proof. (Outline) In one direction, this involves showing that $\approx$ as a relation over configurations satisfies the defining properties of *reduction barbed congruence*. The main problem here is to show that it is contextual, and in particular that $\Pi \models M \approx N$ implies $\Pi \models M|O \approx N|O$ for every O which only has access to the external (accessible) part of Π. This in turn involves developing *Decomposition* and *Composition* theorems for derived actions from configurations of the form $\Pi \triangleright M|O$. The overall structure of the proof is similar to the corresponding result in [8], Proposition 12, but the details are more complicated because of the presence of the network.

The essential part of the converse is to show that for every derived action, relative to a network Π, there is an observer which only uses the external knowledge of Π which completely characterises the effect of that action. These observers have already been constructed for simpler languages such as π-calculus, in [9], and Dπ, in [8]. Here the novelty is to be able to characterise the observable effect that actions have on a network. But it turns out that for every Π we can define an observer O_Π which when run on an arbitrary network Π' can determine whether the external or accessible part of Π' coincides with that of Π.

5 Related Work and Conclusions

We have presented a simple extension of Dπ, in which there is an explicit representation of the underlying network on which processes execute; moreover the network can exhibit both node and link failures. Our main result is a *fully-abstract* bisimulation equivalence with which we can reason about the behaviour of processes in the presence of dynamic network failures. To the best of our knowledge this is the first time system behaviour in the presence of *link* failure has ever been investigated.

Our starting point was the work by Hennessy and Riely [15] on bisimulation techniques for a distributed variant of CCS with location failure. We adapted this work to Dπ and defined a reduction semantics to describe the behaviour of systems in the presence of node and link failures. We then applied techniques of actions dependent on the observer's knowledge, developed for the π-calculusin [9] and Dπ in [8], to characterise a natural notion of barbed congruence.

There has been a number of studies on process behaviour in the presence of *permanent node failure* only, in addition to the already cited [15]. That closest to our work is presented in [3, 2]. They use a located version of the π-calculus, called π_{1l} in which processes are located at explicit sites, which are subject to failure. However processes can not dynamically extend the set of nodes, but more generally π_{1l} is not *location-aware*; in the absence of node failure locations have no impact on computations. Moreover their approach to developing reasoning tools is also quite different from ours. Rather than develop, justify and use bisimulations in the language of interest, π_{1l}, they propose a translation into a version of the π-calculus without locations, and use reasoning tools on the translations. But they do show that for certain π_{1l} terms it is sufficient to reason on their translations. Elsewhere, permanent location failure with hierarchical dependencies have been studied by Fournet, Gonthier, Levy and Remy in [7]. Berger [4] studied a π-calculus extension that models transient location failure with persistent code and communication failures, while Nestmann, Merro and Fuzzatti [14] employ a tailor made process calculus to express standard results in distributed systems, such as [6].

References

1. *Java RMI reference "Java Remote Method Invocation - Distributed Computing for Java" (White Paper)*, May 1999.
2. Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In D. Garlan and D. Le Métayer, editors, *Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION'97)*, volume 1282, pages 374–391, Berlin, Germany, 1997. Springer-Verlag.
3. Roberto M. Amadio and Sanjiva Prasad. Localities and failures. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 14, 1994.
4. Martin Berger. Basic theory of reduction congruence for two timed asynchronous π-calculi. In *Proc. CONCUR'04*, 2004.
5. Luca Cardelli. Wide area computation. In *Proceedings of* 26^{th} *ICALP*, Lecture Notes in Computer Science, pages 10–24. Springer-Verlag, 1999.
6. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
7. Cedric Fournet, Georges Gonthier, Jean Jaques Levy, and Remy Didier. A calculus of mobile agents. *CONCUR 96*, LNCS 1119:406–421, August 1996.
8. Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 322:615–669, 2004.
9. Matthew Hennessy and Julian Rathke. Typed behavioural equivalences for processes in the presence of subtyping. *Mathematical Structures in Computer Science*, 14:651–684, 2004.
10. Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

11. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
12. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
13. Martin W. Murhammer, Orcun Atakan, Stefan Bretz, Larry R. Pugh, Kazunari Suzuki, and David H. Wood. *TCP/IP Tutorial and Technical Overview*. IBM Redbooks. International Technical Support Organization, 6 edition, October 1998.
14. Nestmann, Fuzzati, and Merro. Modeling consensus in a process calculus. In *CONCUR: 14th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2003.
15. James Riely and Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 226:693–735, 2001.