MIKADO
Mobile Calculi based on Domains

MIKADO Global Computing Project
IST-2001-32222

Modal and temporal logics for distributed behaviour

MIKADO Deliverable D2.3.1

Editor : R. CHADHA

Authors : R. CHADHA

Contributors : M. LORETI, R.DE NICOLA (U. FIRENZE)
C. CARREZ AND E. NAJM (ENST)
R. CHADHA, V. SASSONE (U. SUSSEX)

Classification : Public
Deliverable no. : D2.3.1
Reference : RR/WP2/3
Date : January 2004

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

Abstract

This document presents the current results of MIKADO-related work on the development and application of modal and temporal logics for distributed behaviour. The main scientific contents are reported in the technical annexes as the part of the deliverables, and contain the following documents:

1. A modal logic for mobile agents by R. De Nicola and M. Loreti.
2. A distributed Kripke semantics by R. Chadha, D. Macedonio and V. Sassone.
3. Bilogics: Spatial-nominal logics for bigraphs (extended abstract) by G. Conforti, D. Macedonio and V. Sassone.
4. Bigraphical logics for XML by G. Conforti, D. Macedonio and V. Sassone.
5. Assembling components with behavioural contracts by C. Carrez, A. Fantechi, and E. Najm..
6. Formal analysis of multi-party fair exchange protocols by R. Chadha, S. Kremer and A. Scedrov

We provide a brief summary of the scientific contents below.

Work relevant to the deliverable has been carried out at the INRIA, Firenze and Sussex sites. In a wide-area distributed system, both temporal and spatial aspects of systems are important, and the systems are often required to satisfy properties in the presence of phenomena such as untrusted agents and unreliable communication. Several steps are needed to formally analyse whether a system satisfies a given property.

First, a suitable abstraction of the system amenable to analysis has to be identified and formally specified. Models and languages suitable for specifying abstractions of models have been reported in other MIKADO deliverables, such as D.1.2.1, D.1.3.1, D.2.1.2 and D.2.3.2. After the abstraction has been been formally specified, there are different strategies in the literature to verify if the abstraction satisfies the given property. One, which has been reported in MIKADO deliverables D.2.2.1, involves finding an ideal system which obviously satisfies the desired property. The next step is to show that the abstract model behaves like the ideal system. This is indeed the approach of bisimulations that has been discussed in MIKADO deliverables, such as D.2.2.1, D.2.2.2 and D.2.3.2.

Another approach that is often used in the literature is logic-based and involves a) finding a language (syntax) in which desired properties can be written, and b) providing a means for checking whether an abstracted model satisfies this property. A property expressed in the syntax is referred to as *logical formula*, and the means of checking whether an abstracted model satisfies the formula is referred to as *semantics*. It is best to illustrate this strategy by the example of the **Klaim logic** [6] developed in MIKADO and included as part of this deliverable.

In previous MIKADO deliverables, such as D.2.2.1, D.2.2.2, the language Klaim has been reported, and describes a computing paradigm in which processes (called *nets*) and data exist at locations, and can be moved across different locations. The language allows remote operations such as inserting and removing data from remote locations, and spawning a process at a remote location. These operations are captured in terms of a labelled transition system: $N \xrightarrow{a} N'$ represents a net N that performs an operation a, resulting in a net N'. The operation a gives the details of the operation such as the location from which the operation is invoked, the location at which the operation is invoked, and the information transferred.

Now, the basic logical construct in the Klaim logic is of the form

$$\langle A \rangle \psi$$

where A is a *label predicate* and ψ itself is another logical formula. Intuitively, A represents a set of operations, and a net N satisfies $\langle A \rangle \psi$ if there is an operation a in this set such that: $N \xrightarrow{a} N'$ and N' satisfies ψ.

One thing that needs to be pointed out in this logic is that A is not just a syntactic listing of all the operations but a representation of a set, and may contain set operations and references to locations. Furthermore, the logic has a recursion operator. Together, they allow for a rich variety of properties to be expressed such as:

no process at a site different from s_1 can ever retrieve data from site s_2.

There is also a *sound and complete proof system* for the logic considered. A proof system allows new formulae to be derived from others. Soundness here means that if a formula is derived in the proof system, then it will be satisfied for all nets. Completeness means that if a formula is true of all nets then it can be derived in the proof system.

The logical formula $\langle A \rangle \psi$ above is what is often called in the literature a modal formula: the truth of the formula cannot be derived from the truth of its parts. Instead, such formulae have an

intensional meaning. Often these meanings are in terms of possibility or temporality, and are interpreted as *can, could, possibly, may, must, always, and others*. In computer science, the intensional meaning is usually given with the application in mind. For this, a set of models is defined and the validity of a formula is given in terms of a model. In the case of Klaim logic, the nets were the models.

For example, in [3] which is part of the deliverable, the modalities have a spatial interpretation. There are three kinds of modal formulas in the logic considered there; $\psi@p$, $\Diamond\psi$ and $\Box\psi$. Intuitively, a model satisfies $\psi@p$ if there is a place named p where ψ is true. A model satisfies $\Diamond\psi$ if there is some place where $\Diamond\psi$ is true and satisfies $\Box\psi$ if it is true for all places. The contribution of the report is that the set of models considered form a *sound* and *complete semantics* for the proof system. This logic is useful in reasoning about distribution of resources. For example, one can express properties such as: *if a printer is available at location p, then everybody can access the printer*.

Another logic suitable for reasoning about spatial distribution of resources was developed in [5, 4], as a part of the deliverable. This logic, also known as **BiLogics**, is very expressive, and some other logics which give spatial interpretation to modal formulae can be embedded in this logic. The models of this logic are bigraphs. Bigraphs are a formalism of global computing in which various other formalisms such as π-calculus and ambient calculus can be embedded. Informally, bigraphs are two graph structures that share the same set of nodes. The first graph, *place graph*, is structured as a hierarchical tree and describes the placement of nodes. The second, *link graph*, is a hyper graph and formalises the connections between these nodes.

In [5], embedding of two modal logics with spatial interpretation is given. For example, they demonstrate an embedding of a logic, Propositional Spatial Tree Logic (STL), which is interpreted over forests of trees whose nodes are labelled. The basic modal formula in STL is of the form $a[\psi]$, and a forest satisfies $a[\psi]$ if it contains exactly one tree whose root is labelled with a and the forest of the sub-trees satisfy ψ. A fragment of BiLogics suitable for describing and reasoning about XML is identified in [4].

Another application of modal logics was explored in [2] which is part of the deliverable. In this report, a modal logic is used to study contract-signing protocols which allow multiple signers to exchange digitally signed contracts. The logic is interpreted over a game-based model. The basic modal formula of the logic is of the $\langle P\rangle\psi$, meaning that the player P has a strategy to achieve ψ. The properties desired of contract-signing protocols are complicated, and are naturally stated in terms of strategies. For example, the basic desired property of contract signing is fairness, which can be stated as: *if the signer P_1 gets P_2's contract, then P_2 should have a strategy to get P_1's contract*.

Hence, the logical construct $\langle P\rangle\psi$ allows the properties of contract-signing protocols to be stated formally. In [2], two multi-party protocols are analysed using a finite-state tool. The analysis revealed several errors in one protocol, and amendments to the protocol were proposed in order to fix them.

Another application of modal logics was explored in [1] which is part of the deliverable. Here, the use of modal logics is very different from the general plan described above. Modal formulae appear as types, and well-typedness ensures certain safety guarantees. The use of logical formulae as types is well-studied in the literature, and is often referred to as *propositions-as-types, proofs-as-programs* paradigm. The type system given in this report is for component interfacing which we elaborate on now.

Component based design is an emerging paradigm for building distributed systems, where

a system is built using interconnected components. The compatibility of a component with its environment has to guaranteed before its deployment. In the report, this is achieved by using *typed interfaces* for components.

The components themselves are abstracted as a set of ports together with a set of internal threads. The components communicate with its environment using its ports, and the actions of threads effect the ports. Each component can exhibit several typed interfaces which are used for communicating with its environment (*i.e.* other components). The types of interfaces restrict the sequence of actions to be performed by the interface, and uses modalities. There are two kinds of modalities in the types: the **must** modality specifies the required messages and the **may** modality specifies the possible ones. The types are called *behavioral contracts*, and a definition of a *component honoring a contract* is defined. Furthermore, a system made of several connected components is shown to satisfy well-typedness and liveliness if all the components honor the contracts given by their interfaces, and the communicating interfaces are compatible.

This summarises the work done in MIKADO relevant to this deliverable. As we have seen, the work is quite extensive and addresses both theoretical and practical aspects.

References

[1] C. Carrez, A. Fantechi, and E. Najm. Assembling components with behavioural contracts. *Annales des Télécommunications*, 2005. to appear.

[2] R. Chadha, S. Kremer, and A. Scedrov. Formal analysis of multi-party fair exchange protocols. In *17th IEEE Computer Security Foundations Workshop*, pages 266–279, Asilomar, CA, USA, 2004. IEEE Computer Society Press.

[3] R. Chadha, D. Macedonio, and V. Sassone. A distributed Kripke semantics. Technical report 2004:4, Department of Informatics, University of Sussex, Brighton, UK, 2004.

[4] G. Conforti, D. Macedonio, and V. Sassone. Bigraphical logics for XML. Unpubilshed notes. `http://www.di.unipi.it/~confor/publications.html`, October 2004.

[5] G. Conforti, D. Macedonio, and V. Sassone. Bilogics: Spatial-nominal logics for bigraphs (extended abstract). `http://www.di.unipi.it/~confor/publications.html`, October 2004.

[6] R. De Nicola and M. Loreti. A modal logic for mobile agents. *ACM Trans. Comput. Logic*, 5(1):79–128, 2004.

A Modal Logic for Mobile Agents

ROCCO DE NICOLA and MICHELE LORETI
Università di Firenze

KLAIM is an experimental programming language that supports a programming paradigm where both processes and data can be moved across different computing environments. The language relies on the use of explicit localities. This paper presents a temporal logic for specifying properties of Klaim programs. The logic is inspired by Hennessy-Milner Logic (HML) and the $\mu-$calculus, but has novel features that permit dealing with state properties and impact of actions and movements over the different sites. The logic is equipped with a complete proof system that enables one to prove properties of mobile systems.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; *Semantics*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Modal logic*; *Temporal logic*

General Terms: Languages, Security, Theory, Verification

Additional Key Words and Phrases: Mobile Code Languages, Temporal Logics of Programs, Coordination Models, Proof Systems, Logics, Mobility

1. INTRODUCTION

The increasing use of wide area networks, especially the Internet, has stimulated the introduction of new programming paradigms and languages that model interactions among hosts by means of *mobile agents*; these are programs that are transported and executed on different sites. For this class of programs, like for other formalisms, it is crucial to have tools for establishing deadlock freeness, liveness, and correctness with respect to given specifications. However for programs involving different actors and authorities it is also important to establish other properties such as resource allocation, access to resources, and information disclosure. For this purpose, in this paper we advocate the

This work has been partially supported by EU within the FET-Global Computing Initiative, project MIKADO IST-2001-32222 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.
Authors' address: Dipartimento di Sistemi e Informatica, Università di Firenze, Via Lombroso, 6/17, 50134 Firenze, Italy; email: {denicola,loreti}@dsi.unifi.it.

use of temporal logics for specifying and verifying dynamic properties of mobile agents running over a wide area network.

Modal logics have been largely used as formal tools for specifying and verifying properties of concurrent systems. Properties are specified by means of *temporal* and *spatial modalities*. To verify whether a concurrent system satisfies a formula two main approaches are classically available. These are respectively based on *proof systems* and *model checking*. In the proof system approach, inference rules are provided in order to build proofs that establish the satisfaction of formulae by systems. In the model checking approach, a methodology is introduced to *automatically* verify whether a system belongs to the formal model of a formula. In this paper, we will present a new temporal logic and develop a proof system for it.

We shall first introduce a formalism for describing processes and nets, then we will define a modal logic that fits naturally with the language. Finally, we shall present the proof system. In the rest of this introduction we briefly describe the proposed framework.

KLAIM

KLAIM is based on Linda [Gelernter 1985, 1989] but makes use of *multiple* located tuple spaces and *process operators* inspired by Milner's CCS [Milner 1989].

In Linda, communication is asynchronous and is performed via shared space. Messages are structured and are named *tuples*; the shared space is named *tuple space*. Linda provides two actions for inserting tuples in tuple space (**out** and **eval**) and two actions for retrieving tuples from tuple space (**in** and **read**), where elements are selected by using *pattern matching*.

KLAIM processes are built by using classic CCS operators like parallel composition, nondeterministic choice and action prefixing.

In KLAIM programs, called *nets*, tuple spaces and processes are distributed over different localities $(s, s_1, \ldots)$ and the classical Linda operations are indexed with the location of the tuple space they operate on to obtain $\mathbf{out}(t)@s$, $\mathbf{in}(t)@s$, $\mathbf{read}(t)@s$, and $\mathbf{eval}(P)@s$. This allows programmers to distribute/retrieve data and processes over/from different nodes directly, and thus manage the physical distribution of processes, the allocation policies, and the agent's mobility.

In order to model the evolution of a KLAIM net, we define a labelled operational semantics that, differently from the original one [De Nicola et al. 1998], does not rely on structural congruence and makes use of explicit labels. The labels carry information about the action performed, the localities involved in the action and the transmitted information. Transition labels have the following structure:

$$\mathbf{x}(s_1, arg, s_2)$$

where **x** denotes the kind of the action performed (**o** for **out**, **i** for **in**, **r** for **read** and **e** for **eval**). Locality s_1 denotes the *place* where the action is executed, while s_2 is the locality where the action takes effect. Finally, *arg* is the argument of the action and can be either a tuple or a process. For instance, if from a

process running at locality s_1 inserts a tuple t in the tuple space located at s_2, by executing $\mathbf{out}(t)@s_2$, then the net evolves with a transition whose label is $\mathbf{o}(s_1, t, s_2)$.

A Modal Logic for KLAIM

Since the language is based on process algebras, a natural candidate as a specification formalism is a temporal logic based on HML, the logic proposed by Hennessy and Milner to specify and verify properties of CCS agents [Hennessy and Milner 1985]. However, one soon realizes that HML is not completely satisfactory.

In HML, temporal properties of processes are expressed by means of the *diamond* operator ($\langle A\rangle\phi$) indexed with a set of transition labels. A CCS process P satisfies $\langle A\rangle\phi$ if there exist a label a and a process P' such that $P \stackrel{a}{\rightarrow} P'$ and P' satisfies ϕ.

Since in KLAIM transition labels are more complex and involve many components, the diamond operator will be indexed with a *label predicate*. In the proposed framework, a net N satisfies a formula $\langle\mathcal{A}\rangle\phi$ if there exists a label a and a net N' such that we have: $N \stackrel{a}{\succ\!\!\rightarrow} N', a$ satisfies $\mathcal{A}$ and N' satisfies ϕ. A label predicate $\mathcal{A}$ is built from *basic label predicates* and *abstract actions* by using disjunction ($\cdot \cup \cdot$), conjunction ($\cdot \cap \cdot$) and difference ($\cdot - \cdot$).

Basic label predicates are used for denoting the set of all transition labels ($\circ$) and for referring to labels with specific source (the locality from which the action is performed) and/or target (the locality where the action takes place).

Abstract actions denote set of labels by singling out the kind of action performed (**out**, **in**, ...), the localities involved in the transition and the information transmitted. Abstract actions have the same structure of transition labels; but have *locality references* instead of localities and *tuple and process predicates* instead of tuples and processes. Locality references can be names ($s, s_1, \ldots$) or variables ($u, u_1, u_2, \ldots$). We also use $?u$ to indicate a sort of universal quantification over names. Process and tuple predicates are used for characterizing properties of tuples and processes involved in the transition. For instance, 1_P and 1_t denote generic processes and tuples, respectively. Predicates are introduced in order to describe patterns of tuples and possible computations. For instance,

—$\mathcal{A}_1 = \mathtt{R}(s_1, 1_t, s_2)$ is satisfied by a transition label if a process, located at s_1, retrieves generic tuple from the tuple space at s_2;

—$\mathcal{A}_2 = \mathtt{R}(?u_1, 1_t, s_2)$ is satisfied by a transition label if a process, located at a generic locality, retrieves a generic tuple from the tuple space at s_2;

—$\mathcal{A}_2 - \mathcal{A}_1$ is satisfied by a transition label if a process, that is not located at s_1, retrieves a generic tuple from the tuple space at s_2.

In our logic, there are also state formulae for specifying the distribution of resources (i.e. tuples in tuple spaces) in the system. The logic is equipped with a proof system based on tableau and it is inspired by Cleaveland [1990], Stirling and Walker [1991] and Winskel [1989].

The main differences between our solution and the existing ones, also based on Hennessy-Milner logic, reside on the different transition labels. In the *standard* approaches, even those considering value passing [Rathke and Hennessy 1997], labels are considered *basic entities* and are characterized syntactically inside modal operators. Instead, in our approach transition labels are characterized in terms of their *properties*.

The definition of the proof system is, in some sense, standard. However, since we have an explicit notion of values, proofs of *completeness* and *soundness* are more complex.

A small example: specifying access rights

To show how our logic can be used to formally specify access rights, we present a simple informal description of relative access rights of two nodes.

Let s_1 and s_2 be localities of a net, whose access policies have been set up in a such way that all processes, regardless of their site, can write on s_2 but only processes located at s_1 can read from s_2. Assume we are interested in proving that for this system *"never a process, located at a locality different from s_1, will retrieve a tuple from s_2"*

Using our logic we can formalize this property as follows:

$$\neg\mu\kappa.\langle\mathcal{A}_2 - \mathcal{A}_1\rangle\mathbf{tt} \vee \langle\circ\rangle\kappa$$

where $\mathcal{A}_1$ and $\mathcal{A}_2$ are the predicates defined above, $\circ$ is the basic label predicate denoting any transition label and μ is the recursion operator. The formula $\mu\kappa.\langle\mathcal{A}_2 - \mathcal{A}_1\rangle\mathbf{tt} \vee \langle\circ\rangle\kappa$ states that the system will perform eventually an action that satisfies the predicate $\mathcal{A}_2 - \mathcal{A}_1$. The negation ($\neg$) of the formula specifies that this will never happen and thus that it is never the case that a process which is not located at s_1 reads a tuple located at the tuple space of s_2.

Paper organization

The rest of the paper is organized as follows. Section 2, contains a brief introduction to KLAIM and its labelled operational semantics. The modal logic for KLAIM is presented in Section 3 and Section 4. Section 3 contains syntax and semantics of the proposed logics without recursion; moreover it contains the proof system and the proof of its soundness and completeness. In Section 4, we add recursion to the logic and extend the proof system to the new language establishing again soundness and completeness. Section 5 contains examples of use of our logical framework; we describe a few systems and the specification of some of their key properties. In Section 6, we highlight differences and similarities with other approaches.

2. KLAIM: SYNTAX AND SEMANTICS

KLAIM is a language designed for programming applications over wide area networks. It is based on the notion of *locality* and relies on a Linda-like communication model. Linda [Carriero and Gelernter 1989; Gelernter 1985, 1989] is a coordination language with asynchronous communication and shared memory. The shared space is named *Tuple Space*, tuples are structured data.

Table I. KLAIM Syntactic Categories

Set	Elements	Description
$\mathcal{S}$	s	*Physical localities* or *Sites*
Loc	l	*Logical localities*
$VLoc$	u	*Locality variables*
$VLoc \cup Loc \cup \mathcal{S}$	ℓ	
$VLoc \cup \mathcal{S}$	σ	
Val	v	*Basic values*
Var	x	*Value variables*
Exp	e	*Expressions*
Ψ	A	*Process identifiers*
χ	X	*Process variables*
$\mathcal{R} \subseteq Loc \to \mathcal{S}$	ρ	*Allocation Environments*
Net	N	KLAIM *nets*
$Proc$	P	KLAIM *processes*
Act	act	KLAIM *actions*
$\mathcal{T}$	t	*Tuples*
	et	*Evaluated tuples*
$Subst$	δ	*Substitutions*

Table II. Net Syntax

N	$::=$	$s ::_\rho P$	(node)
	$\mid$	$N_1 \parallel N_2$	(net composition)

In KLAIM, like in Linda, messages are structured and named *tuples*. Tuples are sequences of *actual* and *formal* fields. Actual fields are values. Formal fields are *variables* that will be assigned when a tuple is retrieved. Formal fields are denoted by placing a '!' before the variable. Tuples are retrieved from tuple spaces by pattern matching.

A KLAIM system, called a *net*, is a set of *nodes*, univocally denoted by *physical names*. Every node has a computational component (a set of processes running in parallel) and a data component (a tuple space). The operations over tuple spaces take as argument the name of the node where the target tuple space resides and a tuple. $\mathcal{S}$ denotes the set of *physical names* (or *sites*) while $s, s_1, s_2, \ldots$ denote its elements.

Programs refer to nodes by means of *logical localities*. Logical localities can be thought of as aliases for physical names. The set Loc is used to denote the set of logical localities and $l, l_1, l_2, \ldots$ to denote its elements. The distinct logical locality `self` $\in Loc$ is used to refer the node where programs are running. In KLAIM every node is also equipped with an *allocation environment* (ρ), which maps logical localities to physical localities. Every program binds logical localities to physical localities by using the allocation environment of the node where it is running.

The syntax of KLAIM nets is defined in Table II. A node is defined by three parameters: the physical name s, the allocation environment ρ, and the running process P that also contains information about the local tuple space—the set of tuples residing at s. A net N can be obtained via the parallel composition of nodes.

Table III. Process Syntax

P	::=	**nil**	(null process)
	|	$act.P$	(action prefixing)
	|	**out**(et)	(evaluated tuple process)
	|	$P_1 \mid P_2$	(parallel composition)
	|	$P_1 + P_2$	(nondet. choice)
	|	X	(process variable)
	|	$A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{v} \rangle$	(process invocation)
act	::=	**out**$(t)@\ell$ | **in**$(t)@\ell$ | **read**$(t)@\ell$ | **eval**$(P)@\ell$	
	|	**newloc**(u)	
t	::=	$f \mid f,t$	
f	::=	$v \mid P \mid \ell \mid !x \mid !X \mid !u$	

Table IV. Tuple Evaluation Function

$\mathcal{T}[\![v]\!]_\rho$	=	v	$\mathcal{T}[\![P]\!]_\rho$	=	$P\{\rho\}$	$\mathcal{T}[\![\ell]\!]_\rho$	=	$\rho(\ell)$
$\mathcal{T}[\![!x]\!]_\rho$	=	$!x$	$\mathcal{T}[\![!X]\!]_\rho$	=	$!X$	$\mathcal{T}[\![!u]\!]_\rho$	=	$!u$
			$\mathcal{T}[\![f,t]\!]_\rho$	=	$\mathcal{T}[\![f]\!]_\rho, \mathcal{T}[\![t]\!]_\rho$			

Let *VLoc* and χ be, respectively, sets of locality and process variables, which are ranged over by u and X. Elements of $\mathcal{S} \cup Loc \cup VLoc$ are denoted by ℓ while those of $VLoc \cup \mathcal{S}$ will be denoted by σ. The set of basic values $v, v_1, \ldots$ will be denoted by Val while the set of variables $x, x_1, \ldots$ will be denoted by Var. Here, we do not specify what the basic values are. Finally, Ψ is the set of process identifiers A. The KLAIM syntactic categories are summarized in Table I.

Process syntax is defined in Table III, where **nil** stands for the process that cannot perform any actions, $P_1|P_2$ and $P_1 + P_2$ stand for the parallel and nondeterministic composition of P_1 and P_2, respectively. The term $act.P$ stands for the process that executes the action act then behaves like P. The possible actions are: **out**$(t)@\ell$, **in**$(t)@\ell$, **read**$(t)@\ell$, **eval**$(P)@\ell$ and **newloc**(u).

The **out** action adds the result of evaluating t to the tuple space at site ℓ. A tuple is a sequence of *actual* and *formal* fields. Actual fields are value (v, ℓ or P). Formal fields are obtained by placing '!' before value, locality and process variables ($!x, !u, !X$).

The evaluation rules for tuples are given in Table IV, where $P\{\rho\}$ denotes the closure of P with respect to the allocation environment ρ: the process obtained from P by replacing every occurrence of each logical locality l with the physical locality $\rho(l)$. We will use et for denoting a *fully evaluated* tuple: tuples that, for each allocation environment ρ, $\mathcal{T}[\![et]\!]_\rho = et$.

The actions **in**(t) and **read**(t) are used to retrieve information from tuple spaces. As distinct from **out**, these are blocking operations; the computation is

Table V. The Matching Rules

$match(P, P)$	$match(s, s)$	$match(e, e)$
$match(!X, P)$	$match(!u, s)$	$match(!x, e)$
$match(et_2, et_1)$	$match(et_1, et_2)$	$match(et_3, et_4)$
$match(et_1, et_2)$	$match((et_1, et_3), (et_2, et_4))$	

blocked until the required action can be performed, that is, a tuple matching t can be found.

The matching predicate is defined in Table V. Basically, two values match if they are identical while formal fields match any field of the same type. Two tuples (t_1, t_2) and (t_3, t_4) match if t_1 matches t_3 and t_2 matches t_4. The predicate match is also commutative: if t_1 matches t_2 then t_2 matches t_1. For instance $(1, !x, \mathbf{out}(3)@s.\mathbf{nil})$ matches $(1, 4, !X)$, while $(1, !x, 5)$ does not match $(!x, \mathbf{nil}, s)$.

The $\mathbf{in}(t)@\ell$ action looks for a tuple inside the tuple space at ℓ that satisfies the *matching predicate* defined in Table V. If this tuple et exists then it is removed from the tuple space and the continuation process is closed with respect to the substitution $\delta = [et/t]$ that replaces each formal variable in t with the corresponding value in et. For instance, if $t = (s_1, !x, !X)$ and $et = (!u, 4, \mathbf{out}(3)@s_2.\mathbf{nil})$ then $[t/et] = [4/x, \mathbf{out}(3)@s_2.\mathbf{nil}/X]$.

The **read** operation behaves like **in** but it doesn't remove the tuple. The actions $\mathbf{in}(t)@\ell.P$ and $\mathbf{read}(t)@\ell.P$ act as a binder for variables in the formal fields of t. A variable is *free* if and only if it is not bound. A process P is closed if and only if each variable in P is not *free*.

The primitive $\mathbf{eval}(P)@\ell$ spawns a process P at the site ℓ. The action $\mathbf{newloc}(u)$ creates a new node and binds the variable u to its new/fresh *name* s. Prefix $\mathbf{newloc}(u).P$ binds the locality variable u in P. In the rest of the paper, we shall only consider closed processes.

Process identifiers are used in recursive process definitions. It is assumed that each process identifier A has a single defining equation $A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} P$. All free variables in P are contained in $\widetilde{X}$, $\widetilde{u}$ and $\widetilde{x}$ and all occurrences of process identifiers in P are guarded—under the scope of an action prefixing operator. In order to avoid name clash we will assume that variables used as parameters for process identifiers are not used as *formal field* in any tuples in their definitions.

In KLAIM tuples are modelled as processes; a tuple et is in the tuple space of a node s if and only if in s there is a process $\mathbf{out}(et)$. Moreover it is supposed that $\mathbf{out}(et)$ never appears in the scope of an action prefixing. We will use $\mathcal{T}$ for the set of tuples, *Proc* for the set of KLAIM processes and *Act* for the set of KLAIM actions.

2.1 Operational Semantics

In this section, we present a labelled operational semantics for KLAIM. The original operational semantics [De Nicola et al. 1998] was unlabelled. The

semantics proposed here is labelled and does not rely on structural congruence. This simplifies its use as a model for the logic. The semantics of KLAIM is given in two steps. The first one captures availability of resources and the resources requests put forward by processes. The second one describes the actual use of resources. We shall let $ALab$ and Etl denote the sets of labels a and e defined by the following grammars:

$$a ::= \mathbf{o}(s_1, et, s_2) \mid \mathbf{i}(s_1, et, s_2) \mid \mathbf{r}(s_1, et, s_2) \mid \mathbf{e}(s_1, P, s_2) \mid \mathbf{n}(s_1, -,)s_2$$
$$e ::= a \mid et@s \mid \rho@s$$

We will refer to s_1 as $\mathtt{source}(a)$ and to s_2 as $\mathtt{target}(a)$. The first level of the operational semantics is defined by using the transition $\twoheadrightarrow \subseteq Net \times Etl \times Net$, which is the least relation defined in Table VI. Symmetric rules for $|, +$ and $\|$ have been omitted. To contrast the transition of the first level with those of the second one, we shall call *local* the former and *global* the latter.

Intuitively we have that $N \xrightarrow{et@s} N'$ if there exists a tuple et in the tuple space of s while $N \xrightarrow{\rho@s} N'$ if s is a node of N with allocation environment ρ. Moreover we have a local transition for every basic action. For instance we have a transition:

$$N_1 \xrightarrow{\mathbf{e}(s_1, P, s_2)} N_2$$

if N_1 contains a process, that is located at s_1, which will call for evaluation of P at a site s_2. Analogously, we have a local transition labelled with $\mathbf{o}(s_1, t, s_2)$, $\mathbf{i}(s_1, t, s_2)$ or $\mathbf{r}(s_1, t, s_2)$ whenever there is a process, located at s_1, that wants to perform $\mathbf{out}(t)@s_2$, $\mathbf{in}(t)@s_2$ or $\mathbf{read}(t)@s_2$ respectively.

The **newloc** action is more critical; it requires handling creation of new sites. To this purpose, we shall assume existence of a total ordering over sites and of a function **succ** such that $\mathbf{succ}(s)$ yields the immediate successor of site s within the given total ordering. If N is a net and s a physical locality, we write $s \in N$ if there exists a node s (or a reference to s) in N. We write $s \notin N$ if s is unknown in N. Finally, we define $sup(s, N)$ as follows:

$$sup(s, N) = \begin{cases} s & \text{if } s \notin N \\ sup(\mathbf{succ}(s), N) & \text{otherwise} \end{cases}$$

This function yields the smallest site s' that is not in N and is bigger or equal to s.

Rules (LPARNEWP) and (LPARNEWN) ensure that the name of new node will be fresh. Let us consider $N_1 \parallel N_2$, if

$$N_1 \xrightarrow{\mathbf{n}(s_1, -, s_2)} N_1'$$

and s_2 appears in N_2, a new fresh name s_3 is selected and all the references to s_2 in N_1' are replaced with references to s_3 (the node s_2 does not exists in N'). This permits guaranteeing that $N \xrightarrow{\mathbf{n}(s_1, -, s_2)} N'$ if there exists a node s_1 that would create a new node s_2 that is not in N'.

Notice that, rules (LNEWLOC), (LPARNEWP) and (LPARNEWN) permit expressing the *standard* semantics for KLAIM **newloc** without knowing the

Table VI. The Operational Semantics: Local Transitions

$$(\text{LTuple})\ \frac{}{s ::_\rho \mathbf{out}(et) \xrightarrow{et@s} s ::_\rho \mathbf{nil}} \qquad (\text{LSite})\ \frac{}{s ::_\rho P \xrightarrow{\rho@s} s ::_\rho P}$$

$$(\text{LOut})\ \frac{}{s ::_\rho \mathbf{out}(t)@\ell.P \xrightarrow{\mathbf{o}(s, \mathcal{T}[\![t]\!]_\rho, \rho(\ell))} s ::_\rho P}$$

$$(\text{LEval})\ \frac{}{s ::_\rho \mathbf{eval}(Q)@\ell.P \xrightarrow{\mathbf{e}(s, Q, \rho(\ell))} s ::_\rho P}$$

$$(\text{LIn})\ \frac{}{s ::_\rho \mathbf{in}(t)@\ell.P \xrightarrow{\mathbf{i}(s, \mathcal{T}[\![t]\!]_\rho, \rho(\ell))} s ::_\rho P}$$

$$(\text{LRead})\ \frac{}{s ::_\rho \mathbf{read}(t)@\ell.P \xrightarrow{\mathbf{r}(s, \mathcal{T}[\![t]\!]_\rho, \rho(\ell))} s ::_\rho P}$$

$$(\text{LNewloc})\ \frac{s' = sup(\mathbf{succ}(s), s ::_\rho P')}{s ::_\rho \mathbf{newloc}(u).P \xrightarrow{\mathbf{n}(s, -, s')} s ::_\rho P[s'/u]}$$

$$(\text{LCall})\ \frac{s ::_\rho P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{v}/\tilde{x}] \xrightarrow{a} N'}{s ::_\rho A\langle \tilde{P}, \tilde{\ell}, \tilde{v}\rangle \xrightarrow{a} N'}\ A(\tilde{X}, \tilde{u}, \tilde{x}) \stackrel{def}{=} P$$

$$(\text{LChoice})\ \frac{s ::_\rho P_1 \xrightarrow{a} s ::_\rho P_1'}{s ::_\rho P_1 + P_2 \xrightarrow{a} s ::_\rho P_1'}$$

$$(\text{LParN})\ \frac{N_1 \xrightarrow{a} N_1' \quad a \neq \mathbf{n}(s_1, -, s_2)}{N_1 \parallel N_2 \xrightarrow{a} N_1' \parallel N_2} \qquad (\text{LParP})\ \frac{s_1 ::_\rho P_1 \xrightarrow{a} s_1 ::_\rho P_1' \ a \neq \mathbf{n}(s_1, -, s_2)}{s_1 ::_\rho P_1|P_2 \xrightarrow{a} s_1 ::_\rho P_1'|P_2}$$

$$(\text{LParNewP})\ \frac{s_1 ::_\rho P_1 \xrightarrow{\mathbf{n}(s_1, -, s_2)} s_1 ::_\rho P_1' \quad s_3 = sup(s_2, s_1 ::_\rho P_1|P_2)}{s_1 ::_\rho P_1|P_2 \xrightarrow{\mathbf{n}(s_1, -, s_3)} s_1 ::_\rho P_1'[s_3/s_2]|P_2}$$

$$(\text{LParNewN})\ \frac{N_1 \xrightarrow{\mathbf{n}(s_1, -, s_2)} N_1' \quad s_3 = sup(s_2, N_1 \parallel N_2)}{N_1 \parallel N_2 \xrightarrow{\mathbf{n}(s_1, -, s_3)} N_1'[s_3/s_2] \parallel N_2}$$

composition of all nets. We would like to remark that the same result could be obtained by using a *restriction* operator similar to the one used in π-calculus [Milner et al. 1992]. Our choice has the advantage that it permits dealing with the creation of new names exactly as with other actions and avoids the problems that arise when one has to consider restrictions in the logic.

The second level of the operational semantics is defined as the least relation $\succ\!\!\rightarrow \subseteq Net \times Lab \times Net$ induced by the rules in Table VII. The *global* relation $\succ\!\!\rightarrow$ reflects the intuitive semantics of basic KLAIM actions.

Table VII. The Operational Semantics: Global Transitions

$$(\text{GOUT})\ \frac{N_1 \xrightarrow{\mathbf{o}(s_1,et,s_2)} N_1' \quad N_1' \xrightarrow{\rho@s_2} N_2}{N_1 \succ\!\xrightarrow{\mathbf{o}(s_1,et,s_2)} N_2 \parallel s_2 ::_\rho \mathbf{out}(et)}$$

$$(\text{GIN})\ \frac{N_1 \xrightarrow{\mathbf{i}(s_1,t,s_2)} N_1' \quad N_1' \xrightarrow{et@s_2} N_2 \quad match(t,et)}{N_1 \succ\!\xrightarrow{\mathbf{i}(s_1,et,s_2)} N_2[et/t]}$$

$$(\text{GEVAL})\ \frac{N_1 \xrightarrow{\mathbf{e}(s_1,P,s_2)} N_1' \quad N_1' \xrightarrow{\rho@s_2} N_2}{N_1 \succ\!\xrightarrow{\mathbf{e}(s_1,P,s_2)} N_2 \parallel s_2 ::_\rho P}$$

$$(\text{GREAD})\ \frac{N_1 \xrightarrow{\mathbf{r}(s_1,t,s_2)} N_1' \quad N_1' \xrightarrow{et@s_2} N_2 \quad match(t,et)}{N_1 \succ\!\xrightarrow{\mathbf{r}(s_1,et,s_2)} N_1'[et/t]}$$

$$(\text{GNEWLOC})\ \frac{N_1 \xrightarrow{\mathbf{n}(s_1,-,s_2)} N_1' \quad N_1' \xrightarrow{\rho@s_1} N_2}{N_1 \succ\!\xrightarrow{\mathbf{n}(s_1,-,s_2)} N_2 \parallel s_2 ::_{[s_2/\mathtt{self}]\cdot\rho} \mathbf{nil}}$$

The rule (GOUT) states that, whenever in a net N_1 a process located at s_1 wants to insert an evaluated tuple et in the tuple space located s_2:

$$N_1 \xrightarrow{\mathbf{o}(s_1,et,s_2)} N_1'$$

and, in the net, there exists a node named s_2:

$$N_1' \xrightarrow{\rho@s_2} N_2$$

then the new elementary process $\mathbf{out}(et)$ is placed at s_2:

$$N_1 \succ\!\xrightarrow{\mathbf{o}(s_1,et,s_2)} N_2 \parallel s_2 ::_\rho \mathbf{out}(et)$$

The rule (GEVAL) is very similar to the previous one. In that case, if a process located at s_1 asks for evaluating a process P at s_2:

$$N_1 \xrightarrow{\mathbf{e}(s_1,P,s_2)} N_1'$$

the new process P is activated at s_2:

$$N_1 \succ\!\xrightarrow{\mathbf{e}(s_1,P,s_2)} N_2 \parallel s_2 ::_\rho P$$

The construct **eval** relies on dynamic scoping: the logical localities of P are evaluated using the allocation environment of the destination node ℓ. Conversely, evaluation of process inserted in tuple spaces via **out** relies on static scoping: in P logical localities are evaluated using the allocation environment of the source node.

Rules (GIN) and (GREAD) have the same assumptions but they differ with each other for the conclusion. The first one states that if a process located at s_1 asks for a tuple matching t from the tuple space located at s ($N_1 \xrightarrow{\mathbf{i}(s_1,t,s_2)} N_1'$) and such a tuple et exists in s_2 ($N_1' \xrightarrow{et@s_2} N_2$), then it is retrieved:

$$N_1 \succ\!\!\xrightarrow{\mathbf{i}(s_1,et,s_2)} N_2.$$

In the case of (GREAD), however, the matching tuple is not removed from the tuple space located at s_2 ($N_1 \succ\!\!\xrightarrow{\mathbf{r}(s_1,et,s_2)} N_1'$).

Finally, the rule (GNEWLOC), which handles the creation of new names, states that whenever a process located at s_1 wants to create a new node named s_2:

$$N_1 \xrightarrow{\mathbf{n}(s_1,-,s_2)} N_1'$$

and the allocation environment of s_1 is ρ:

$$N_1' \xrightarrow{\rho@s_1} N_2$$

then a new node named s_2 is added to the net. The allocation environment of the new node is obtained from ρ by binding the logical locality `self` to s_2 ($[s_2/\mathtt{self}]\cdot\rho$):

$$N_1 \succ\!\!\xrightarrow{\mathbf{n}(s_1,-,s_2)} N_2 \parallel s_2 ::_{[s_2/\mathtt{self}]\cdot\rho} \mathbf{nil}$$

We shall now present a few definitions and results that permit guaranteeing finite branching of our operational semantics.

Definition 2.1. Let N and N' be nets we write $N \succ\!\!\longrightarrow^* N'$ if and only if:

—$N' = N$ *or* $\exists a, N'' :\ N \succ\!\!\xrightarrow{a} N'', N'' \succ\!\!\longrightarrow^* N'$.

LEMMA 2.2. *Let N be a net, then $\{N' | \exists e.N \xrightarrow{e} N'\}$ is finite.*

PROOF. The proof goes by induction on the structure of N.

Base of Induction. Let $N = s ::_\rho P$. We prove this case by induction on the structure of P and assume that P_1 and P_2 are such that the set $\{N' | \exists e.s ::_\rho P_i \xrightarrow{e} N'\}$ is finite.

—$P = \mathbf{nil}$: $\{N' | \exists e.s ::_\rho \mathbf{nil} \xrightarrow{e} N'\}) = \{s ::_\rho \mathbf{nil}\}$

—$P = \mathbf{out}(et)$: $\{N' | \exists e.s ::_\rho \mathbf{out}(et) \xrightarrow{e} N'\} = \{s ::_\rho \mathbf{out}(et), s ::_\rho \mathbf{nil}\}$

—$P = act.P_1$: $\{N' | \exists e.s ::_\rho act.P \xrightarrow{e} N'\} = \{s ::_\rho P', s ::_\rho act.P_1\}$ where, let $s' = sup(\mathbf{succ}(s), s ::_\rho P)$, $P' = P_1[s/u]$ if $act = \mathbf{newloc}(u)$ and $P' = P_1$ otherwise.

—$P = P_1|P_2$:

$$\begin{aligned}\{N' | \exists e.s ::_\rho P_1|P_2 \xrightarrow{e} N'\} = {} & \{s ::_\rho P_1'|P_2 | \exists e.s ::_\rho P_1 \xrightarrow{e} s ::_\rho P_1'\} \\ & \cup \{s ::_\rho P_1|P_2' | \exists e.s ::_\rho P_2 \xrightarrow{e} s ::_\rho P_2'\} \\ & \cup \{s ::_\rho P_1|P_2\}\end{aligned}$$

which is finite for inductive hypothesis.

—$P = P_1 + P_2$:

$$\{N' | \exists e.s ::_\rho P_1 + P_2 \xrightarrow{e} \gg N'\} = \{s ::_\rho P_1' | \exists e.s ::_\rho P_1 \xrightarrow{e} \gg s ::_\rho P_1'\} \cup \{s ::_\rho P_2' | \exists e.s ::_\rho P_2 \xrightarrow{e} \gg s ::_\rho P_2'\}$$

which is finite for inductive hypothesis.

—$P = A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{v} \rangle$. Let $A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} P'$. Since every occurrence of A is guarded in P', this case reduces to one of those considered above.

Inductive Hypothesis. Let N_1 and N_2 be such that following set is finite:

$$\{N' | \exists a.N_i \xrightarrow{a} \gg N'\}$$

Inductive Step. Let $N = N_1 \parallel N_2$. We have that the set of possible next nets is:

$$\{N_1' \parallel N_2 | \exists a.N_1 \xrightarrow{e} \gg N_1'\} \cup \{N_1 \parallel N_2' | \exists a.N_2 \xrightarrow{e} \gg N_2'\},$$

which is finite for inductive hypothesis. □

THEOREM 2.3. *Let N be a net, then $\{N' | \exists a.N \succ\!\!\xrightarrow{a} N'\}$ is finite.*

PROOF. The claim follows directly from the above lemma. □

2.2 A Small Example: Itinerant Agent

In this section, we present the evolution of a simple net. Let N be the net defined as follows:

$$s_1 :: Proc_1 | \mathbf{out}(s_2) \parallel s_2 :: \mathbf{nil}$$

where

$$Proc_1 \stackrel{def}{=} \mathbf{in}(!u)@\mathtt{self}.\mathbf{eval}(Proc_1)@u.\mathbf{out}(\mathtt{self})@u.\mathbf{nil}$$

This process, located at s_1, retrieves a locality at `self` (**in**$(!u)$@`self`). Later, the process evaluates itself and inserts the locality `self` in the tuple space of the read locality (**eval**$(Proc_1)$@u.**out**(`self`)@u.**nil**).

The proposed net evolves according to the operational semantics of Tables VI–VII as follows:

(1) First the tuple (s_2) is retrieved from the tuple space of s_1. With a transition labelled $\mathbf{i}(s_1, (s_2), s_1)$ the net evolves in
$$N_1 = s_1 :: \mathbf{eval}(Proc_1)@s_2.\mathbf{out}(\mathtt{self})@s_2.\mathbf{nil} \parallel s_2 :: \mathbf{nil}.$$

(2) The process $Proc_1$ is evaluated at s_2. The net evolves with a transition labelled $\mathbf{e}(s_1, Proc_1, s_2)$ in the net
$$N_2 = s_1 :: \mathbf{out}(\mathtt{self})@s_2.\mathbf{nil} \parallel s_2 :: Proc_1.$$

(3) Tuple (s_1) is inserted in the tuple space of s_2 transition labelled $\mathbf{o}(s_1, s_1, s_2)$ the net evolves in
$$N_3 = s_1 :: \mathbf{nil} \parallel s_2 :: Proc_1 | \mathbf{out}(s_1).$$

(4) Tuple (s_1) is retrieved from s_2, and with a transition labelled $\mathbf{i}(s_2, (s_1), s_2)$ the net evolves in
$$N_4 = s_1 :: \mathbf{nil} \parallel s_2 :: \mathbf{eval}(Proc_1)@s_1.\mathbf{out}(\mathtt{self})@s_1.\mathbf{nil}.$$

(5) The process $Proc_1$ is evaluated from s_2 at s_1. By a transition labelled $\mathbf{e}(s_2, Proc_1, s_1)$ the net evolves in:

$$N_5 = s_1 :: Proc_1 \parallel s_2 :: \mathbf{out}(\mathtt{self})@s_1.\mathbf{nil}.$$

(6) Finally, with a transition labelled $\mathbf{o}(s_2, (s_2), s_1)$ net N_5 evolves in N.

3. A MODAL LOGIC FOR KLAIM: THE fiNITE FRAGMENT

In this section, we introduce a logic that allows us to specify and prove properties of a mobile system specified in KLAIM. Our approach draws inspiration from Hennessy-Milner Logic [Hennessy and Milner 1985]. In their logical framework, dynamic properties of CCS processes are captured by making use of modal operators, $\langle\rangle$ and [], indexed over a set of labels denoting basic actions of processes.

As we have seen, in KLAIM labels are not basic entities but carry complex information like processes, tuples, sites, and so on. In our logic, the diamond operator $\langle\cdot\rangle$ is indexed with predicates that will be interpreted over labels, and used to specify subsets of the set of possible labels. Using this approach, we shall be able to finitely refer to an infinite set of labels. To characterize state properties of KLAIM programs, namely the presence of a tuple in a specific tuple space, we shall also introduce specific state formulae.

In this section we shall concentrate on the finite fragment of our logic, the next section will be dedicated to formalizing our results for the logic with recursion.

Syntax

We shall let $\mathcal{L}$ be the set of formulas ϕ induced by the following grammar:

$$\phi \ ::= \ \mathtt{tp}@\sigma \ \big| \ \mathbf{tt} \ \big| \ \langle\mathcal{A}\rangle\phi \ \big| \ \phi \vee \phi \ \big| \ \neg\phi$$

State formulae $\mathtt{tp}@\sigma$, where σ denotes either a physical locality or a locality variable (see Table I), are used to specify the distribution of resources over the net (tuples over the different sites). A net N satisfies $\mathtt{tp}@s$ if there is a tuple *et* in the tuple space located at s that *satisfies* $\mathtt{tp}$.

As usual **tt** is the formula satisfied by every net, $\phi_1 \vee \phi_2$ is the formula satisfied by all nets that satisfy either ϕ_1 or ϕ_2 while $\neg\phi$ is satisfied by every net that does not satisfy ϕ. Finally, to satisfy $\langle\mathcal{A}\rangle\phi$ a net N needs to evolve, by an action satisfying the predicate $\mathcal{A}$, in a net that satisfies ϕ.

Semantics

We shall introduce the interpretation function $\mathbb{M}[\![\cdot]\!] : \mathcal{L} \to 2^{Net}$ that, for each $\phi \in \mathcal{L}$, yields the set of nets that satisfy ϕ or, equivalently, the set of nets that are *models* for ϕ.

The function $\mathbb{M}[\![\cdot]\!]$ is defined by relying on functions $\mathbb{A}[\![\cdot]\!]$ and $\mathbb{T}[\![\cdot]\!]$ that provide the interpretations of label and tuple predicates. For each label predicate function $\mathbb{A}[\![\cdot]\!]$ yields a set of pairs *(transition label,substitution)* . $\mathbb{T}[\![\ \mathtt{tp}]\!]$, instead, denotes the set of tuples that satisfy $\mathtt{tp}$. Both $\mathbb{A}[\![\cdot]\!]$ and $\mathbb{T}[\![\cdot]\!]$ will be formally defined later.

Table VIII. Label Predicate Syntax

$\mathcal{A}$	::=		(label predicates)	α	::=		(abstract actions)
		$\circ$	(all labels)			$\mathtt{O}(\mathtt{sr}_1, \mathtt{tp}, \mathtt{sr}_2)$	
	¦	$\mathtt{Src}(\widetilde{\mathtt{sr}})$	(source)		¦	$\mathtt{I}(\mathtt{sr}_1, \mathtt{tp}, \mathtt{sr}_2)$	
	¦	$\mathtt{Trg}(\widetilde{\mathtt{sr}})$	(target)		¦	$\mathtt{R}(\mathtt{sr}_1, \mathtt{tp}, \mathtt{sr}_2)$	
	¦	α	(abstract actions)		¦	$\mathtt{E}(\mathtt{sr}_1, \mathtt{pp}, \mathtt{sr}_2)$	
	¦	$\mathcal{A}_1 \cap \mathcal{A}_2$	(conjunction)		¦	$\mathtt{N}(\mathtt{sr}_1, -, \mathtt{sr}_2)$	
	¦	$\mathcal{A}_1 \cup \mathcal{A}_2$	(disjunction)	sr	::=		(site references)
	¦	$\mathcal{A}_1 - \mathcal{A}_2$	(difference)			s	(physical locality)
					¦	$?u$	(quantified variable)
					¦	u	(free variable)

Let ϕ be a closed formula; $\mathbb{M}[\![\cdot]\!]$ is defined inductively on the structure of ϕ as follows:

—$\mathbb{M}[\![\mathbf{tt}]\!] = Net$;
—$\mathbb{M}[\![\ \mathtt{tp}@\sigma\ ]\!] = \{N | \exists et.N \xrightarrow{et@\sigma} N', et \in \mathbb{T}[\![\ \mathtt{tp}]\!]\}$;
—$\mathbb{M}[\![\langle\mathcal{A}\rangle\phi]\!] = \{N | \exists a, \delta, N' : N \succ\!\!\xrightarrow{a} N',\ (a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!],\ N' \in \mathbb{M}[\![\phi\{\delta\}]\!]\}$;
—$\mathbb{M}[\![\phi_1 \vee \phi_2]\!] = \mathbb{M}[\![\phi_1]\!] \cup \mathbb{M}[\![\phi_2]\!]$;
—$\mathbb{M}[\![\neg\phi]\!] = Net - \mathbb{M}[\![\phi]\!]$;

The interpretation function respects the intuitive semantics of formulae. Every net is a *model* for **tt** while every net with a node s, whose tuple space contains a tuple t satisfying $\mathtt{tp}$, is a model for $\mathtt{tp}@s$. N is a model for $\langle\mathcal{A}\rangle\phi$ if there exist N' such that $N \succ\!\!\xrightarrow{a} N', (a, \delta) \in \mathbb{A}[\![\mathcal{A}]\!]$ and $N' \in \mathbb{M}[\![\phi\{\delta\}]\!]$. Finally, N is a model for $\phi_1 \vee \phi_2$ if N is a model for ϕ_1 or ϕ_2 while N is a model for $\neg\phi$ if it is not a model for ϕ.

We write $N \models \phi$ if and only if $N \in \mathbb{M}[\![\phi]\!]$. Other formulae like $[\mathcal{A}]\phi$ or $\phi_1 \wedge \phi_2$ can be expressed in $\mathcal{L}$. Indeed $[\mathcal{A}]\phi = \neg\langle\mathcal{A}\rangle\neg\phi$ and $\phi_1 \wedge \phi_2 = \neg(\phi_1 \vee \phi_2)$. We shall use these derivable formulae as *macros* in $\mathcal{L}$.

In $\langle\mathcal{A}\rangle\phi$ the operator $\langle\mathcal{A}\rangle$ acts as a binder for the (locality) variables in ϕ that appear quantified in $\mathcal{A}$. As usual u is free in ϕ if there exists an instance of u in ϕ that is not bound and ϕ is closed if it does not contain free variables. Finally, the closure of ϕ with respect to substitution δ (written $\phi\{\delta\}$) is the formula ϕ' obtained from ϕ by replacing every free variable u with $\delta(u)$.

3.1 Label Predicates

In this section, we present a formal syntax and semantics of label predicates. These permit denoting an infinite set of labels by specifying their properties. The set $\mathtt{PLab}$ of label predicates ($\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2, \ldots$) is defined in Table VIII.

Table IX. Label Predicates Interpretation

$\mathbb{A}[\![\circ]\!] = Lab$
$\mathbb{A}[\![\mathtt{O}(\mathtt{sr}_1,\ \mathtt{tp},\ \mathtt{sr}_2)]\!] = \{(\mathbf{o}(s_1,t,s_2);\delta_1\cdot\delta_2)\mid(s_1;\delta_1)\in\mathbb{R}[\![\mathtt{sr}_1]\!],(s_2;\delta_2)\in\mathbb{R}[\![\mathtt{sr}_2]\!],t\in\mathbb{T}[\![\mathtt{tp}]\!]\}$
$\mathbb{A}[\![\mathtt{I}(\mathtt{sr}_1,\ \mathtt{tp},\ \mathtt{sr}_2)]\!] = \{(\mathbf{i}(s_1,t,s_2);\delta_1\cdot\delta_2)\mid(s_1;\delta_1)\in\mathbb{R}[\![\mathtt{sr}_1]\!],(s_2;\delta_2)\in\mathbb{R}[\![\mathtt{sr}_2]\!],t\in\mathbb{T}[\![\mathtt{tp}]\!]\}$
$\mathbb{A}[\![\mathtt{R}(\mathtt{sr}_1,\ \mathtt{tp},\ \mathtt{sr}_2)]\!] = \{(\mathbf{r}(s_1,t,s_2);\delta_1\cdot\delta_2)\mid(s_1;\delta_1)\in\mathbb{R}[\![\mathtt{sr}_1]\!],(s_2;\delta_2)\in\mathbb{R}[\![\mathtt{sr}_2]\!],t\in\mathbb{T}[\![\mathtt{tp}]\!]\}$
$\mathbb{A}[\![\mathtt{E}(\mathtt{sr}_1,\mathtt{pp},\ \mathtt{sr}_2)]\!] = \{(\mathbf{e}(s_1,P,s_2);\delta_1\cdot\delta_2)\mid(s_1;\delta_1)\in\mathbb{R}[\![\mathtt{sr}_1]\!],(s_2;\delta_2)\in\mathbb{R}[\![\mathtt{sr}_2]\!],P\in\mathbb{P}[\![\mathtt{pp}]\!]\}$
$\mathbb{A}[\![\mathtt{N}(\mathtt{sr}_1,-,\ \mathtt{sr}_2)]\!] = \{(\mathbf{n}(s_1,-,s_2);\delta_1\cdot\delta_2)\mid(s_1;\delta_1)\in\mathbb{R}[\![\mathtt{sr}_1]\!],(s_2;\delta_2)\in\mathbb{R}[\![\mathtt{sr}_2]\!]\}$
$\mathbb{A}[\![\mathtt{Src}(\widetilde{\mathtt{sr}})]\!] = \{(a;\emptyset)\mid\exists\delta.(\mathtt{source}(a),\delta)\in\bigcup_{\mathtt{sr}'\in[\![\mathtt{SR}]\!]}\mathbb{R}[\![\mathtt{sr}]\!].\}$
$\mathbb{A}[\![\mathtt{Trg}(\widetilde{\mathtt{sr}})]\!] = \{(a;\emptyset)\mid\exists\delta.(\mathtt{target}(a),\delta)\in\bigcup_{\mathtt{sr}'\in[\![\mathtt{SR}]\!]}\mathbb{R}[\![\mathtt{sr}]\!].\}$
$\mathbb{A}[\![\mathcal{A}_1\cup\mathcal{A}_1]\!] = \mathbb{A}[\![\mathcal{A}_1]\!]\cup\mathbb{A}[\![\mathcal{A}_2]\!]$
$\mathbb{A}[\![\mathcal{A}_1\cap\mathcal{A}_1]\!] = \{(a;\delta_1\cdot\delta_2)\mid(a;\delta_1)\in\mathbb{A}[\![\mathcal{A}_1]\!],(a;\delta_2)\mathbb{A}[\![\mathcal{A}_2]\!]\}$
$\mathbb{A}[\![\mathcal{A}_1-\mathcal{A}_2]\!] = \{(a;\delta)\mid(a;\delta)\in\mathbb{A}[\![\mathcal{A}_1]\!],\forall\delta'\ (a;\delta')\notin\mathbb{A}[\![\mathcal{A}_2]\!]\}$

A label predicate $\mathcal{A}$ is built from *basic label predicates* and *abstract actions* by using disjunction ($\cdot\cup\cdot$), conjunction ($\cdot\cap\cdot$) and difference ($\cdot-\cdot$).

Basic label predicates are used for denoting the set of all transition labels ($\circ$) and for selecting labels basing the choice on their source ($\mathtt{Src}(\widetilde{\mathtt{sr}})$)—the locality from which the action is performed—or on their target ($\mathtt{Src}(\widetilde{\mathtt{sr}})$)—the locality where the action takes place.

Abstract actions (α) denote a set of labels by singling out the kind (`O` for **out**, `I` for **in**, etc.), the localities involved in the transition and the information transmitted. Abstract actions have the same structure as transition labels; however, there are *locality references* (`sr`) instead of localities and *tuple and process predicates* instead of tuples and processes. Locality references can be names ($s, s_1, \ldots$), free variables ($u, u_1, \ldots$) or quantified variables($?u, ?u_1, \ldots$). Process and tuple predicates are used for characterizing properties of tuples and processes involved in the transition.

Semantics. Formal interpretation of label predicates is defined by means of two interpretation functions: $\mathbb{R}[\![\cdot]\!]$ and $\mathbb{A}[\![\cdot]\!]$. The first takes as argument a locality reference `sr` (s, u or $?u$) and yields a set of pairs $<$*physical locality-substitution*$>$. The second takes a label predicate $\mathcal{A}$ and yields a set of pairs $<$*transition label-substitution*$>$. Intuitively, $(a,\delta)\in\mathbb{A}[\![\mathcal{A}]\!]$ if transition label a *satisfies* $\mathcal{A}$ with respect to the substitution δ.

Definition 3.1. We define the interpretation of `sr`, written $\mathbb{R}[\![\mathtt{sr}]\!]$, as follows:

$$\mathbb{R}[\![?u]\!] = \{(s;[s/u])\mid s\in Sites\} \qquad \mathbb{R}[\![u]\!] = \{(u;\emptyset)\} \qquad \mathbb{R}[\![s]\!] = \{(s;\emptyset)\}$$

Notice that, $?u$ is used like a *wild card* that can be replaced by every physical locality. This reference plays the role of *existential quantification* if it is

Table X. Label Predicate Bound Variables

$\mathtt{bv}(\circ) = \emptyset$
$\mathtt{bv}(\mathtt{Src}(\overline{\mathtt{sr}})) = \mathtt{bv}(\mathtt{Trg}(\overline{\mathtt{sr}})) = \bigcup_{\mathtt{sr}\in\overline{\mathtt{sr}}} \mathtt{bv}(\mathtt{sr})$
$\mathtt{bv}(\mathtt{O}(\mathtt{sr}_1,\ \mathtt{tp},\ \mathtt{sr}_2)) = \mathtt{bv}(\mathtt{I}(\mathtt{sr}_1,\ \mathtt{tp},\ \mathtt{sr}_2)) = \mathtt{bv}(\mathtt{R}(\mathtt{sr}_1,\ \mathtt{tp},\ \mathtt{sr}_2)) = \mathtt{bv}(\mathtt{sr}_1) \cup \mathtt{bv}(\mathtt{sr}_2)$
$\mathtt{bv}(\mathtt{E}(\mathtt{sr}_1, \mathtt{pp}, \mathtt{sr}_2)) = \mathtt{bv}(\mathtt{sr}_1) \cup \mathtt{bv}(\mathtt{sr}_2)$
$\mathtt{bv}(\mathbf{n}(\mathtt{sr}_1, -, \mathtt{sr}_2)) = \mathtt{bv}(\mathtt{sr}_1) \cup \mathtt{bv}(\mathtt{sr}_2)$
$\mathtt{bv}(\mathcal{A}_1 \cup \mathcal{A}_2) = \mathtt{bv}(\mathcal{A}_1) \cap \mathtt{bv}(\mathcal{A}2)$
$\mathtt{bv}(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathtt{bv}(\mathcal{A}_1) \cup \mathtt{bv}(\mathcal{A}_2)$
$\mathtt{bv}(\mathcal{A}_1 - \mathcal{A}_2) = \mathtt{bv}(\mathcal{A}_1)$
$\mathtt{bv}(?u) = \{u\}$
$\mathtt{bv}(s) = \mathtt{bv}(u) = \emptyset$

used inside a $\langle\cdot\rangle$, while it works like a *universal quantification* when used instead $[\cdot]$.

Moreover, in $\langle\mathcal{A}\rangle\phi$, $\mathcal{A}$ acts as binder for variables in $\mathcal{A}$. Let be $\mathcal{A}$ a label predicate; the set of variables $\mathtt{bv}(\mathcal{A})$ that are bound in $\mathcal{A}$ is defined in Table X.

Definition 3.2. The interpretation of label predicates is defined in Table IX. Function $\mathbb{A}[\![\cdot]\!]$ relies on $\mathbb{P}[\![\cdot]\!]$ and $\mathbb{T}[\![\cdot]\!]$ that yield, respectively, the set of processes and of tuples satisfying process and tuple predicates.

As an example, consider the label predicates presented in the Introduction. We have:

—$\mathbb{A}[\![\mathtt{R}(s_1, 1_\mathtt{t}, s_2)]\!] = \{(\mathbf{r}(s_1, t, s_2), \emptyset) | t \in \mathcal{T}\}$
—$\mathbb{A}[\![\mathtt{R}(?u_1, 1_\mathtt{t}, s_2)]\!] = \{(\mathbf{r}(s, t, s_2), [s/u_1]) | s \in \mathcal{S}, t \in \mathcal{T}\}$
—$\mathbb{A}[\![\mathtt{R}(?u_1, 1_\mathtt{t}, s_2) - \mathtt{I}(s_1, 1_\mathtt{t}, s_2)]\!] = \{(\mathbf{r}(s, t, s_2), [s/u_1]) | s \in \mathcal{S}, s \neq s_1, t \in \mathcal{T}\}$

Within our logic, label predicates can be used to express spatial properties: *where* actions take place.

3.2 Process and Tuple Predicates

When defining label predicates, we made the use of both tuple and process predicates. These are used to characterize properties of tuples and processes involved in the actions.

Process predicates shall be used to specify the kind of accesses to the resources of the net (tuples and sites). These accesses are composed for specifying their causal dependencies. The causal properties we intend to express for processes are those expressing properties like *"first read something and then use the acquired information in some way"*. Tuple predicates will characterize tuple pattern by specifying the relevant properties of their fields.

We shall use `PTuple` to denote the set of *tuple predicates* `tp` defined as follows:

tp	::=		**(tuple predicate)**
		1_t	(generic tuple)
	|	vp	(value predicate)
	|	lp	(locality predicate)
	|	pp	(process predicate)
	|	$!u$	(locality formal)
	|	$!x$	(value formal)
	|	$!X$	(process formal)
	|	tp, tp	(composition)

The tuple predicate 1_t is used for denoting a generic tuple (any tuple will satisfy 1_t). Value (vp), locality (lp) and process predicates (pp) are used for characterizing the properties of the values that compose the tuples, while formals ($!u$, $!x$ and $!X$) are used to denote formal fields of tuples. Finally, tuple predicates are composed by using comma (tp_1, tp_2). We will have that t_1, t_2 satisfies tp_1, tp_2 if t_1 satisfies tp_1 and t_2 satisfies tp_2.

A *locality predicate* lp is defined as follows:

lp	::=		**(locality predicate)**
		1_s	(generic locality)
	|	s	(physical locality)
	|	u	(locality variable)
	|	l	(logical locality)

where 1_s is used for a generic locality, s and l refer to specific physical or logical locality, respectively, and u is a locality variable.

The *value predicates* vp are defined similarly

vp	::=		**(value predicate)**
	|	1_v	(generic value)
	|	v	(basic value)
	|	x	(value variable)

we have terms for denoting: any value (1_v), a specific value v or a free variable u.

We shall use PLoc and PVal, respectively, for denoting the set of locality predicates and the set of value predicates.

Finally, we define the sets of process and access predicates PProc and PAcc like the set of terms pp and ap defined by the following grammar:

pp	::=		**(process predicate)**
		$1_{\mathtt{P}}$	(generic process)
	|	$\mathtt{ap} \to \mathtt{pp}$	(access requirements)
	|	$\mathtt{pp} \wedge \mathtt{pp}$	(conjunction)
	|	X	(process variable)
ap	::=		**(access predicate)**
		$\mathtt{i}(\mathtt{tp})@\mathtt{lp}$	**(in)**
	|	$\mathtt{r}(\mathtt{tp})@\mathtt{lp}$	**(read)**
	|	$\mathtt{o}(\mathtt{tp})@\mathtt{lp}$	**(out)**
	|	$\mathtt{e}(\mathtt{pp})@\mathtt{lp}$	**(eval)**
	|	$\mathtt{n}(u)$	**(newloc)**

We use $1_{\mathtt{P}}$ for a generic process, $\mathtt{pp}_1 \wedge \mathtt{pp}_2$ for the set of processes that *satisfy* $\mathtt{pp}_1$ and $\mathtt{pp}_2$. A process satisfies $\mathtt{ap} \to \mathtt{pp}$ if it might perform an access (i.e. an action) that satisfies ap and use the acquired information as specified by pp. We introduce an action predicate ap for every basic KLAIM action.

We will write $\mathtt{ap}_1 =_\alpha \mathtt{ap}_2$ if ap is equal to $\mathtt{ap}'$ after changing the name of the variables in their formal fields ($\alpha-$renaming). If $\mathtt{ap}_1 =_\alpha \mathtt{ap}_2$, we will use $[\mathtt{ap}_2/\mathtt{ap}_1]$, for the substitution of every *formal* variable of $\mathtt{ap}_1$ with the corresponding one of $\mathtt{ap}_2$: for instance $\mathtt{r}(!u_1, P, !X_1)@1_{\mathtt{s}} =_\alpha \mathtt{r}(!u_2, P, !X_2)@1_{\mathtt{s}}$ and $[\mathtt{r}(!u_2, P, !X_2)@1_{\mathtt{s}}/\mathtt{r}(!u_1, P, !X_1)@1_{\mathtt{s}}] = [u_2/u_1, X_2/X_1]$.

In process predicates of the form $\mathtt{ap} \to \mathtt{pp}$ the terms $\mathtt{i}(\mathtt{tp})@\mathtt{lp}$, $\mathtt{r}(\mathtt{tp})@\mathtt{lp}$ and $\mathtt{n}(u)$ act as a binder for the *formal* variables in tp and for the variable u. A variable is free in pp (respectively tp) if it does not appear under the scope of a binder $\mathtt{i}(\mathtt{tp})@\mathtt{lp}$, $\mathtt{r}(\mathtt{tp})@\mathtt{lp}$ and $\mathtt{n}(u)$. We will use $\mathtt{fv}(\mathtt{pp})$ to denote the set of free variables in pp. Function $\mathtt{fv}(\cdot)$ is formally defined in Table XI, where $\mathtt{form}(\mathtt{tp})$ yields the set of variables that are used like formals in the tuple predicate tp.

Process predicates can be thought as types that reflect the possible accesses a process might perform along its computation; they also carry information about the possible use of the acquired resources [De Nicola et al. 2000]. This allows us to specify sophisticated properties on process capabilities. For instance we can specify the set of processes that, after reading the name of a locality from u_1, spawn a process to the read locality:

$$\mathtt{i}(!u)@s_1 \to \mathtt{e}(1_{\mathtt{P}})@u \to 1_{\mathtt{P}}$$

This predicate is satisfied by both:

$$\mathbf{in}(!u_1)@s_1.\mathbf{eval}(P)@u_1.Q$$

$$\mathbf{in}(!u_1)@s_1.\mathbf{in}(!x)@u_1.\mathbf{eval}(P)@u_1.Q$$

but it is not satisfied by

$$\mathbf{in}(!u_2)@s_1.\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.\mathbf{nil}$$

Table XI. Free Variables of Process Predicates

$$\mathtt{form}(1_{\mathtt{t}}) = \mathtt{form}(\,\mathtt{vp}) = \mathtt{form}(\mathtt{pp}) = \mathtt{form}(\,\mathtt{lp}) = \emptyset$$
$$\mathtt{form}(!x) = \{x\}$$
$$\mathtt{form}(!X) = \{X\}$$
$$\mathtt{form}(!u) = \{u\}$$
$$\mathtt{form}(\,\mathtt{tp}_1,\,\mathtt{tp}_2) = \mathtt{form}(\,\mathtt{tp}_1) \cup \mathtt{form}(\,\mathtt{tp}_2)$$

$$\mathtt{fv}(1_{\mathtt{t}}) = \mathtt{fv}(!u) = \mathtt{fv}(!x) = \mathtt{fv}(!X) = \emptyset$$
$$\mathtt{fv}(\,\mathtt{tp}_1,\,\mathtt{tp}_2) = \mathtt{fv}(\,\mathtt{tp}_1) \cup \mathtt{fv}(\,\mathtt{tp}_2)$$
$$\mathtt{fv}(1_{\mathtt{p}}) = \emptyset$$
$$\mathtt{fv}(\mathtt{pp}_1 \wedge \mathtt{pp}_2) = \mathtt{fv}(\mathtt{pp}_1) \cup \mathtt{fv}(\mathtt{pp}_2)$$
$$\mathtt{fv}(\mathtt{ap} \rightarrow \mathtt{pp}) = (\mathtt{fv}(\mathtt{pp}) - \mathtt{bv}(\mathtt{ap})) \cup \mathtt{fv}(\mathtt{ap})$$
$$\mathtt{fv}(X) = \{X\}$$
$$\mathtt{fv}(\mathtt{o}(\,\mathtt{tp})@\,\mathtt{lp}) = \mathtt{fv}(\mathtt{i}(\,\mathtt{tp})@\,\mathtt{lp}) = \mathtt{fv}(\mathtt{r}(\,\mathtt{tp})@\,\mathtt{lp}) = \mathtt{fv}(\,\mathtt{tp}) \cup \mathtt{fv}(\,\mathtt{lp})$$
$$\mathtt{fv}(\mathtt{e}(\mathtt{pp})@\,\mathtt{lp}) = \mathtt{fv}(\mathtt{pp}) \cup \mathtt{fv}(\,\mathtt{lp})$$
$$\mathtt{fv}(\mathtt{n}(u)@\,\mathtt{lp}) = \emptyset$$
$$\mathtt{fv}(x) = \{x\}$$
$$\mathtt{fv}(u) = \{u\}$$
$$\mathtt{fv}(1_{\mathtt{v}}) = \mathtt{fv}(v) = \mathtt{fv}(1_{\mathtt{s}}) = \mathtt{fv}(s) = \mathtt{fv}(l) = \emptyset$$

$$\mathtt{bv}(\mathtt{i}(\,\mathtt{tp})@\,\mathtt{lp}) = \mathtt{bv}(\mathtt{r}(\,\mathtt{tp})@\,\mathtt{lp}) = \mathtt{form}(\,\mathtt{tp})$$
$$\mathtt{bv}(\mathtt{o}(\,\mathtt{tp})@\,\mathtt{lp}) = \mathtt{bv}(\mathtt{e}(\mathtt{pp})@\,\mathtt{lp}) = \emptyset$$
$$\mathtt{bv}(\mathtt{n}(u)) = \{u\}$$

Table XII. Interpretation for Predicates

$\mathbb{L}[\![1_{\mathtt{s}}]\!] = \mathcal{S}$	$\mathbb{L}[\![s]\!] = \{s\}$	$\mathbb{L}[\![l]\!] = \{l\}$	$\mathbb{L}[\![u]\!] = \{u\}$
$\mathbb{V}[\![1_{\mathtt{v}}]\!] = Val$	$\mathbb{V}[\![v]\!] = \{v\}$	$\mathbb{V}[\![x]\!] = \{x\}$	
$\mathbb{T}[\![1_{\mathtt{t}}]\!] = \mathcal{T}$	$\mathbb{T}[\![\,\mathtt{lp}]\!] = \mathbb{L}[\![\,\mathtt{lp}]\!]$	$\mathbb{T}[\![\,\mathtt{vp}]\!] = \mathbb{V}[\![\,\mathtt{vp}]\!]$	$\mathbb{T}[\![\mathtt{pp}]\!] = \mathbb{P}[\![\mathtt{pp}]\!]$
$\mathbb{T}[\![!u]\!] = \{!u\}$	$\mathbb{T}[\![!x]\!] = \{!x\}$	$\mathbb{T}[\![!X]\!] = \{!X\}$	

indeed, in the latter case, no process is evaluated at the locality retrieved from s_1.

3.2.1 *Semantics.* In this section, we show how the predicates introduced in the previous section are interpreted.

Interpretation of locality, value and tuples predicates are, in some sense, obvious. The formal definition of functions

$$\mathbb{L}[\![\cdot]\!] : \mathtt{PLoc} \rightarrow \mathcal{S} \cup Loc \cup VLoc$$
$$\mathbb{V}[\![\cdot]\!] : \mathtt{PVal} \rightarrow Val \cup Var$$
$$\mathbb{T}[\![\cdot]\!] : \mathtt{PTuple} \rightarrow \mathcal{T}$$

can be found in Table XII. The syntactic categories are those of Table I.

To define $\mathbb{P}[\![]\!]$ we need to introduce a transition relation for describing possible computations of processes. Relations $\twoheadrightarrow$ and $\rightarrowtail$ introduced in the previous section are not adequate, because they describe the actual computation of nets and processes. The relation we need, instead, has to describe, using a sort of *abstract interpretation*, what actions a process might perform during

its computation. Let $\mathcal{V}$ be a set of variables, we will write $P \stackrel{act}{\Longrightarrow_{\mathcal{V}}} Q$ if:

—the process P, at some point of its computation, might perform the action act;

—all the actions that are executed before act, do not bind variables in $\mathcal{V}$.

Definition 3.3. Let $\mathcal{V} \subset VLoc \cup \chi \cup Var$ the relation $\rightarrow_{\mathcal{V}} \subset Proc \times Proc$ is defined as follows:

—if act does not bind variables in $\mathcal{V}$ then

$$act P \rightarrow_{\mathcal{V}} P$$

—for every P and Q

$$P|Q \rightarrow_{\mathcal{V}} P$$
$$P|Q \rightarrow_{\mathcal{V}} Q$$
$$P + Q \rightarrow_{\mathcal{V}} P$$
$$P + Q \rightarrow_{\mathcal{V}} Q$$

—if $A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} P$ then

$$A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{v} \rangle \rightarrow_{\mathcal{V}} P[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}]$$

The relation $\stackrel{}{\Longrightarrow_{\mathcal{V}}} \subset Proc \times Act \times Proc$ is defined as follows:

—for every $\mathcal{V}$,

$$act.P \stackrel{act}{\Longrightarrow_{\mathcal{V}}} P$$

—if $P \rightarrow_{\mathcal{V}} P'$ and $P' \stackrel{act}{\Longrightarrow_{\mathcal{V}}} Q$ then

$$P \stackrel{act}{\Longrightarrow_{\mathcal{V}}} Q$$

Functions $\mathbb{P}[\![\cdot]\!] : \mathtt{PProc} \rightarrow Proc$ and $\mathbb{AC}[\![\cdot]\!] : \mathtt{PAcc} \rightarrow Act$ are defined as follows:

$$\mathbb{P}[\![\mathtt{1_P}]\!] = Proc$$

$$\mathbb{P}[\![X]\!] = \{X\}$$

$$\mathbb{P}[\![\mathtt{ap} \rightarrow \mathtt{pp}]\!] = \{P | \exists act, Q, \mathtt{ap}' \quad \mathtt{ap} =_{\alpha} \mathtt{ap}', act \in \mathbb{P}[\![\mathtt{ap}']\!], Q \in \mathbb{P}[\![\mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]]\!], P \stackrel{act}{\Longrightarrow_{\mathtt{fv}(\mathtt{ap} \rightarrow \mathtt{pp})}} Q\}$$

$$\mathbb{P}[\![\mathtt{pp}_1 \wedge \mathtt{pp}_2]\!] = \mathbb{P}[\![\mathtt{pp}_1]\!] \cap \mathbb{P}[\![\mathtt{pp}_2]\!]$$

$$\mathbb{AC}[\![\mathtt{o(\,tp)@\,lp}]\!] = \{(\mathbf{out}(t)@\ell | t \in \mathbb{P}[\![\,\mathtt{tp}]\!],\ \ell \in \mathbb{P}[\![\,\mathtt{lp}]\!]\}$$

$$\mathbb{AC}[\![\mathtt{i(\,tp)@\,lp}]\!] = \{(\mathbf{in}(t)@\ell | t \in \mathbb{P}[\![\,\mathtt{tp}]\!],\ \ell \in \mathbb{P}[\![\,\mathtt{lp}]\!]\}$$

$$\mathbb{AC}[\![\mathtt{r(\,tp)@\,lp}]\!] = \{(\mathbf{read}(t)@\ell | t \in \mathbb{P}[\![\,\mathtt{tp}]\!],\ \ell \in \mathbb{P}[\![\,\mathtt{lp}]\!]\}$$

$$\mathbb{AC}[\![\mathtt{e(pp)@\,lp}]\!] = \{\mathbf{eval}(Q)@\ell | \ell \in \mathbb{P}[\![\,\mathtt{lp}]\!],\ Q \in \mathbb{P}[\![\mathtt{pp}]\!]\sigma\}$$

$$\mathbb{AC}[\![\mathtt{n}(u)]\!] = \{(\mathbf{newloc}(u') | u' \in VLoc\}$$

We will write $P : \mathtt{pp}, t : \mathtt{tp}$ and $act : \mathtt{ap}$ if $P \in \mathbb{P}[\![\mathtt{pp}]\!], t \in \mathbb{T}[\![\mathtt{tp}]\!]$ and $act \in \mathbb{P}[\![\mathtt{ap}]\!]$ respectively. Conversely, we will write $\neg(P : \mathtt{pp})$, $\neg(t : \mathtt{tp})$ and $\neg(act : \mathtt{ap})$ if $P \notin \mathbb{P}[\![\mathtt{pp}]\!], t \notin \mathbb{P}[\![\mathtt{tp}]\!]$ and $act \notin \mathbb{P}[\![\mathtt{ap}]\!]$. Furthermore, we assume that process predicates are equal up to contraction (i.e. $\mathtt{pp} \wedge \mathtt{pp} = \mathtt{pp}$), commutative and associative properties; for instance $\mathtt{pp}_1 \wedge (\mathtt{pp}_2 \wedge \mathtt{pp}_1) = \mathtt{pp}_1 \wedge \mathtt{pp}_2$.

If we consider predicate $\mathtt{pp} = \mathtt{i}(!u)@s_1 \rightarrow \mathtt{e}(1_\mathtt{P})@u \rightarrow 1_\mathtt{P}$, introduced in the previous Section, we have that:

$$\mathbf{in}(!u_1)@s_1.\mathbf{in}(!x)@u_1.\mathbf{eval}(P)@u_1.Q : \mathtt{pp}$$

indeed:

$$\mathbf{in}(!u_1)@s_1.\mathbf{in}(!x)@u_1.\mathbf{eval}(P)@u_1.Q \xrightarrow[\emptyset]{\mathbf{in}!u_1@s_1} \mathbf{in}(!x)@u_1.\mathbf{eval}(P)@u_1.Q \xrightarrow[\{u_1\}]{\mathbf{eval}(P)@u_1} Q$$

Moreover:

$\mathbf{in}(!u_1)@s_1 : \mathtt{i}(!u_1)@s_1 =_\alpha \mathtt{i}!u@s_1$,
$\mathtt{e}(1_\mathtt{P})@u[\mathtt{i}(!u_1)@s_1/\mathtt{i}!u@s_1] = \mathtt{e}(1_\mathtt{P})@u_1$,
$\mathbf{eval}(P)@u_1 : \mathtt{e}1_\mathtt{P}@u_1$ and $Q : 1_\mathtt{P}$,
$\mathbf{in}(!u_1)@s_1.\mathbf{in}(!x)@u_1.\mathbf{eval}(P)@u_1.Q : \mathtt{pp}$

On the contrary, the process

$$\mathbf{in}(!u_2)@s_1.\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.\mathbf{nil}$$

does not satisfy the predicate pp, indeed:
$\mathbf{in}(!u_2)@s_1.\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.\mathbf{nil} \xrightarrow[\emptyset]{\mathbf{in}(!u_2)@s_1} \mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.\mathbf{nil}$

Even if:

$$\mathbf{in}(!u_2)@s_1 : \mathtt{i}(!u_2)@s_1 =_\alpha \mathtt{i}(!u)@s_1$$

process $\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.\mathbf{nil}$ does not satisfy $\mathtt{e}(1_\mathtt{P})@u_2 \rightarrow 1_\mathtt{P}$. Indeed:

—the action $\mathbf{read}(!u_2)@s_2$ binds the variable u_2;
—there is no process P' such that $\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.\mathbf{nil} \rightarrow_{u_2} P'$.

Hence, $\mathbf{in}(!u_2)@s_1.\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.\mathbf{nil} \rightarrow_{u_2} P'$ does not satisfy pp.

We would like to remark that process predicates represent a *set* of *causal dependent* sequences of *accesses* that a single process *might* perform and not actual computational sequences. Thus process predicates are not able to distinguish $P_1|P_2$ from $P_1 + P_2$. Indeed, in $P_1|P_2$, like in $P_1 + P_2$, the accesses that P_1 might perform are not *dependent* on those that P_2 might perform. In that sense, $P_1|P_2$ and $P_1 + P_2$ satisfy the same set of predicates.

3.2.2 *A Proof System for Process and Tuple Predicates.* In this section, we will introduce a proof system that permits verifying the satisfiability of a predicate by a process or a tuple. For the proof system we will prove soundness and completeness.

Table XIII. The Proof System for Process and Tuple Predicates

Process rules:

$$\Delta \vdash P : \mathtt{1_P} \qquad \Delta, X : \mathtt{pp} \vdash X : \mathtt{pp} \qquad \Delta \vdash X : X \ (1)$$

$$\mathtt{Rw}\ \frac{\Delta, X : \mathtt{pp}_i \vdash P : \mathtt{pp}}{\Delta, X : \mathtt{pp}_1 \wedge \mathtt{pp}_2 \vdash P : \mathtt{pp}}\ i \in \{1,2\} \qquad \mathtt{R}\wedge\ \frac{\Delta \vdash P : \mathtt{pp}_1 \quad \Delta \vdash P : \mathtt{pp}_2}{\Delta \vdash P : \mathtt{pp}_1 \wedge \mathtt{pp}_2}$$

$$\mathtt{R}|\ \frac{\Delta \vdash P_i : \mathtt{ap} \to \mathtt{pp}}{\Delta \vdash P_1 | P_2 : \mathtt{ap} \to \mathtt{pp}}\ i \in \{1,2\} \qquad \mathtt{R}+\ \frac{\Delta \vdash P_i : \mathtt{ap} \to \mathtt{pp}}{\Delta \vdash P_1 + P_2 : \mathtt{ap} \to \mathtt{pp}}\ i \in \{1,2\}$$

$$\mathtt{RAct}_1\ \frac{\Delta \vdash P : \mathtt{ap} \to \mathtt{pp}}{\Delta \vdash act.P : \mathtt{ap} \to \mathtt{pp}}\ (2)$$

$$\mathtt{RAct}_2\ \frac{\Delta \vdash act : \mathtt{ap}' \quad \Delta \vdash P : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]}{\Delta \vdash act.P : \mathtt{ap} \to \mathtt{pp}}\ (3)$$

$$\mathtt{RCall}\ \frac{X_1 : \mathtt{pp}_1, \ldots, X_n : \mathtt{pp}_n \vdash P'[\tilde{\ell}/\tilde{u}, \tilde{v}/\tilde{x}] : \mathtt{ap} \to \mathtt{pp} \qquad \Delta \vdash P_1 : \mathtt{pp}_1 \ldots \Delta \vdash P_n : \mathtt{pp}_n}{\Delta \vdash A\langle P_1, \ldots, P_n, \tilde{\ell}, \tilde{v}\rangle : \mathtt{ap} \to \mathtt{pp}}\ (4)$$

Action rules:

$$\mathtt{ROut}\ \frac{\Delta \vdash t : \mathtt{tp} \quad \Delta \vdash \ell : \mathtt{lp}}{\Delta \vdash \mathbf{out}(t)@\ell : \mathtt{o(\,tp)@\,lp}} \qquad \mathtt{RRead}\ \frac{\Delta \vdash t : \mathtt{tp} \quad \Delta \vdash \ell : \mathtt{lp}}{\Delta \vdash \mathbf{read}(t)@\ell : \mathtt{r(\,tp)@\,lp}}$$

$$\mathtt{RIn}\ \frac{\Delta \vdash t : \mathtt{tp} \quad \Delta \vdash \ell : \mathtt{lp}}{\Delta \vdash \mathbf{in}(t)@\ell : \mathtt{i(\,tp)@\,lp}} \qquad \mathtt{REval}\ \frac{\Delta \vdash P : \mathtt{pp} \quad \Delta \vdash \ell : \mathtt{lp}}{\Delta \vdash \mathbf{eval}(P)@\ell : \mathtt{e(pp)@\,lp}}$$

$$\mathtt{RNew}\ \Delta \vdash \mathbf{newloc}(u) : \mathtt{n}(u)$$

Tuple rules:

$$\Delta \vdash !u : !u \qquad \Delta \vdash !X : !X \qquad \Delta \vdash v : v \qquad \Delta \vdash v : \mathtt{1_v} \qquad \Delta \vdash \ell : \ell \qquad \Delta \vdash \ell : \mathtt{1_s} \qquad \Delta \vdash t : \mathtt{1_t}$$

$$\frac{\Delta \vdash t_1 : \mathtt{tp}_1 \quad \Delta \vdash t_2 : \mathtt{tp}_2}{\Delta \vdash t_1, t_2 : \mathtt{tp}_1, \mathtt{tp}_2}\ \mathtt{RTuple}$$

(1) X does not appear in Δ
(2) act does not bind free variables in $\mathtt{ap} \to \mathtt{pp}$
(3) $\mathtt{ap} =_\alpha \mathtt{ap}'$
(4) $A(X_1, \ldots, X_n, \tilde{u}, \tilde{x}) \stackrel{def}{=} P'$ and $\forall\, i.\ \mathtt{pp}_i \in (\downarrow \mathtt{ap} \to \mathtt{pp})^C$

Definition 3.4. *Process, action* and *tuple sequents* are of the form:

$$\Delta \vdash P : \mathtt{pp}\ ,\ \Delta \vdash act : \mathtt{ap}\ ,\ \Delta \vdash t : \mathtt{tp}$$

where *the assumptions* Δ are sets of pairs <*process variable,process predicate*>.

In order to limit the size of the set of predicates that can appear in a proof, we introduce the operators $\downarrow \cdot$ and $\Downarrow \cdot$.

Definition 3.5. Let pp be a process predicate and let $\{\widetilde{\mathtt{pp}}\}$ be a set of process predicates, we define $\downarrow \mathtt{pp}$ and $\Downarrow \{\widetilde{\mathtt{pp}}\}$ as follows:

—$\downarrow 1_{\mathtt{P}} = \{1_{\mathtt{P}}\}$;
—$\downarrow (\mathtt{pp}_1 \wedge \mathtt{pp}_2) = \downarrow \mathtt{pp}_1 \cup \downarrow \mathtt{pp}_2 \cup (\downarrow \mathtt{pp}_1) \wedge (\downarrow \mathtt{pp}_2)$
—$\downarrow (\mathtt{ap} \rightarrow \mathtt{pp}) = \downarrow \mathtt{pp} \cup \downarrow \mathtt{ap} \cup (\mathtt{ap} \rightarrow \downarrow \mathtt{pp})$

and

—$\Downarrow \emptyset = \emptyset$;
—$\Downarrow \{\mathtt{pp}\} \cup \{\widetilde{\mathtt{pp}}\} = (\downarrow \mathtt{pp}) \cup (\Downarrow \{\widetilde{pp}\}) \cup ((\downarrow \mathtt{pp}) \wedge (\Downarrow \{\widetilde{\mathtt{pp}}\}))$

where:

$$(\downarrow \mathtt{pp}_1) \wedge (\downarrow \mathtt{pp}_2) = \{\mathtt{pp}' \wedge \mathtt{pp}" | \mathtt{pp}' \in \downarrow \mathtt{pp}_1 \text{ and } \mathtt{pp}" \in \downarrow \mathtt{pp}_2\}$$
$$\downarrow \mathtt{ap} = \Downarrow \{\mathtt{pp} | \mathtt{pp} \in \mathtt{ap}\}$$
$$(\mathtt{ap} \rightarrow \downarrow \mathtt{pp}) = \{\mathtt{ap} \rightarrow \mathtt{pp}' | \mathtt{pp}' \in \downarrow \mathtt{pp}\}$$
$$\downarrow \mathtt{tp} = \Downarrow \{\mathtt{pp} | \mathtt{pp} \in \mathtt{tp}\}$$

Let $\{\mathtt{pp}\}$ be a set of process predicates; we use $(\{\widetilde{\mathtt{pp}}\})^C$ for the $\wedge$-closure of $\{\mathtt{pp}\}$.

The proof system for process and tuples predicates is defined in Table XIII. Three groups of rules are considered: *process rules*, *action rules* and *tuple rules*. The more interesting are *process rules* while action and tuple rules are, in some sense, standard.

Process rules contain three axioms which, respectively, state that: every process satisfies $1_{\mathtt{P}}$; if it is assumed that X satisfies $\mathtt{pp}$ then $X : \mathtt{pp}$ is provable; and every process variable X *satisfies itself*. The condition "X *does not appear in* Δ on axiom $\Delta \vdash X : X$ is used to avoid name clash between formal variables and parameter variables in process definitions. There is a *weakening rule* (Rw): if one is able to prove $P : \mathtt{pp}$ under $\Delta \cup \{X : \mathtt{pp}_1\}$ then $P : \mathtt{pp}$ is provable under $\Delta \cup \{X : \mathtt{pp}_1 \wedge \mathtt{pp}_2\}$. The rule for predicate conjunction (R$\wedge$) ($\mathtt{pp}_1 \wedge \mathtt{pp}_2$) is as usual: $P : \mathtt{pp}_1 \wedge \mathtt{pp}_2$ under Δ if both $P : \mathtt{pp}_1$ and $P : \mathtt{pp}_2$ under Δ. Parallel composition and nondeterministic choice are treated in the proof system like *disjunctions*. There exists a proof for $\Delta \vdash P_1 | P_2$ or $\Delta \vdash P_1 + P_2$ if there exists a proof either for $\Delta \vdash P_1 : \mathtt{pp}$ or for $\Delta \vdash P_2 : \mathtt{pp}$ (rules R|and R+). Two rules can be applied to prove $\Delta \vdash act.P : \mathtt{ap} \rightarrow \mathtt{pp}$ ($\mathtt{RAct}_1$and $\mathtt{RAct}_2$). If act does not bind free variables in $\mathtt{ap} \rightarrow \mathtt{pp}$ and there exists a proof for $\Delta \vdash P : \mathtt{ap} \rightarrow \mathtt{pp}$ then there exists a proof for $\Delta \vdash act.P : \mathtt{ap} \rightarrow \mathtt{pp}$. On the other hand, if there exists a proof for $\Delta \vdash act : \mathtt{ap}', \mathtt{ap}' =_\alpha \mathtt{ap}$ and there exists a proof for $\Delta \vdash P : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]$ then there exists a proof for $\Delta \vdash act.P : \mathtt{ap} \rightarrow \mathtt{pp}$. Finally, one is able to prove $A\langle P_1, \ldots, P_n, \widetilde{\ell}, \widetilde{v}\rangle$, where $A(X_1, \ldots, X_n) \stackrel{def}{=} P'$, if there are $\mathtt{pp}_1, \ldots, \mathtt{pp}_n \in (\downarrow \mathtt{pp})^C$ such that there exist proofs for $X_1 : \mathtt{pp}_1, \ldots, X_n : \mathtt{pp}_x \vdash P'[\widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}]$, $\Delta \vdash P_1 : \mathtt{pp}_1, \ldots, \Delta \vdash P_1 : \mathtt{pp}_1$ (RCall).

Previously, we showed that process $\mathbf{in}(!u_1)@s_1.\mathbf{eval}(P)@u_1.Q$ belongs to the interpretation of $\mathtt{i}(!u)@s_1 \rightarrow \mathtt{e}(1_{\mathtt{P}})@u \rightarrow 1_{\mathtt{P}}$. Using the proof system, we can build, for the sequent $\vdash \mathbf{in}(!u_1)@s_1.\mathbf{eval}(P)@u_1.Q : \mathtt{i}(!u)@s_1 \rightarrow \mathtt{e}(1_{\mathtt{P}})@u \rightarrow 1_{\mathtt{P}}$,

the following proof:

$$\dfrac{\dfrac{\vdash !u_1 : !u_1 \quad \vdash s_1 : s_1}{\vdash \mathbf{in}(!u_1)@s_1 : \mathtt{i}(!u_1)@s_1}\,\mathtt{RIn} \qquad \dfrac{\dfrac{\vdash P : \mathtt{1_P} \quad \vdash u_1 : u_1}{\vdash \mathbf{eval}(P)@u_1 : \mathtt{e}(\mathtt{1_P})@u_1}\,\mathtt{REval} \qquad \vdash Q : \mathtt{1_P}}{\vdash \mathbf{eval}(P)@u_1.Q : \mathtt{e}(\mathtt{1_P})@u_1 \to \mathtt{1_P}}\,\mathtt{RAct_2}}{\vdash \mathbf{in}(!u_1)@s_1.\mathbf{eval}(P)@u_1.Q : \mathtt{i}(!u)@s_1 \to \mathtt{e}(\mathtt{1_P})@u \to \mathtt{1_P}}\,\mathtt{RAct_2}$$

Now we are ready to present soundness and completeness results.

THEOREM 3.6 (SOUNDNESS). *For every process P, action act tuple t, predicates pp, ap and tp and $\Delta = X_1 : pp_1, \ldots, X_n : pp_n$, $\Delta \vdash P : pp$, $\Delta \vdash act : ap$ and $\Delta \vdash t : tp$ are provable then for every process $P_1, \ldots, P_n$ such that, for every i, $P_i : pp_i$, then $P[P_1/X_1, \ldots, P_n/X_n] : pp$, $act[P_1/X_1, \ldots, P_n/X_n] : ap$ and $t[P_1/X_1, \ldots, P_n/X_n] : tp$.*

PROOF. We prove, by induction on the number of steps needed to prove $\Delta \vdash P : \mathtt{pp}$, $\Delta \vdash act : \mathtt{ap}$ and $\Delta \vdash t : \mathtt{tp}$, that if there exists a proof for $\Delta \vdash P : \mathtt{pp}$, $\Delta \vdash act : \mathtt{ap}$ and $\Delta \vdash t : \mathtt{tp}$ then for every process $P_1, \ldots, P_n$ such that, for every i, $P_i : \mathtt{pp}_i$, then $P[P_1/X_1, \ldots, P_n/X_n] : \mathtt{pp}$, $act[P_1/X_1, \ldots, P_n/X_n] : \mathtt{ap}$ and $t[P_1/X_1, \ldots, P_n/X_n] : \mathtt{tp}$.

Base. If $\Delta \vdash P : \mathtt{pp}$, $\Delta \vdash act : \mathtt{ap}$ and $\Delta \vdash t : \mathtt{tp}$ are axioms, i.e. provable in 0 steps, the following cases can be distinguished:

— $\Delta \cup \{X_i : \mathtt{pp}_i\} \vdash X_i : \mathtt{pp}_1$, $\Delta \vdash P : \mathtt{1_P}$ or $\Delta \vdash X : X$ then the statement follows easily, indeed: in the first case, $P_i : \mathtt{pp}_i$ for hypothesis; in the second one for each $P_1, \ldots, P_n$, $P[P_1/X_1, \ldots, P_n/X_n]$ satisfies $\mathtt{1_P}$ and in the latter, since X does not appear in Δ, $X[P_1/X_1, \ldots, P_n/X_n] = X$ and $X : X$;

— $\Delta \vdash v : \mathtt{1_v}$, $\Delta \vdash v : v$ $\Delta \vdash !u : !u$, $\Delta \vdash !X : !X$, $\Delta \vdash \ell : \mathtt{1_s}$, $\Delta \vdash \ell : \ell$ and $\Delta \vdash t : \mathtt{1_t}$. For each of these case, the statement follows easily.

Inductive Hypothesis. For every $X_1 : \mathtt{pp}_1, \ldots, X_i : \mathtt{pp}_i$, if $X_1 : \mathtt{pp}_1, \ldots, X_i \vdash P : \mathtt{pp}$, $X_1 : \mathtt{pp}_1, \ldots, X_i :\vdash act : \mathtt{ap}$ and $X_1 : \mathtt{pp}_1, \ldots, X_i \vdash t : \mathtt{tp}$ are provable within n steps then for every $P_1 : \mathtt{pp}_1, \ldots, P_i : \mathtt{pp}_i$: $P[P_1/X_1, \ldots, P_i/X_i] : \mathtt{pp}$, $act[P_1/X_1, \ldots, P_i/X_i] : \mathtt{ap}$ and $t[P_1/X_1, \ldots, P_i/X_i] : \mathtt{tp}$.

Inductive Step. Let us assume that $\Delta \vdash P : \mathtt{pp}$, $\Delta \vdash act : \mathtt{ap}$ and $\Delta \vdash t : \mathtt{tp}$ are provable within $n + 1$ steps. We have to distinguish according to the last applied rule.

(Rw). In that case, the proof starts:

$$\dfrac{\{X_1 : \mathtt{pp}_1^1, \ldots, X_i : \mathtt{pp}_i\} \vdash P : \mathtt{pp}}{\{X : \mathtt{pp}_1^1 \wedge \mathtt{pp}_1^2, \ldots, X_i : \mathtt{pp}_i\} \vdash P : \mathtt{pp}}$$

Since $\{X_1 : \mathtt{pp}_1^1, \ldots, X_i : \mathtt{pp}_i\} \vdash P : \mathtt{pp}$ is provable within n steps, then:

$$\forall\, P_1 : \mathtt{pp}_1^1, \ldots, P_i : \mathtt{pp}_i \Rightarrow P[P_1/X_1, \ldots, P_i/X_i] : \mathtt{pp}$$

Moreover, for each P if $P : \mathtt{pp}_1 \wedge \mathtt{pp}_2$ then $P : \mathtt{pp}_1$. Hence:

$$\forall\, P_1 : \mathtt{pp}_1^1 \wedge \mathtt{pp}_1^2, \ldots, P_i : \mathtt{pp}_i \Rightarrow P[P_1/X_1, \ldots, P_i/X_i] : \mathtt{pp}$$

(R$\wedge$). the statement follows easily from the inductive hypothesis and by definition of $\mathbb{P}[\![\mathtt{pp}_1 \wedge \mathtt{pp}_2]\!] = \mathbb{P}[\![\mathtt{pp}_1]\!] \cap \mathbb{P}[\![\mathtt{pp}_2]\!]$;

(R|and R+). also in that case the statement follows by inductive hypothesis and by noting that, for each P_1 and P_2, if $P_1 : \mathtt{ap} \to \mathtt{pp}$ then both $P_1|P_2$ and $P_1 + P_2$ belong to $\mathbb{P}[\![\mathtt{ap} \to \mathtt{pp}]\!]$;

($\mathtt{RAct}_1$). the proof starts:

$$\frac{\{X_1 : \mathtt{pp}_1, \ldots, X_i : \mathtt{pp}_i\} \vdash P : \mathtt{ap}_1 \to \mathtt{pp}_1}{\{X_1 : \mathtt{pp}_1, \ldots, X_i : \mathtt{pp}_i\} \vdash act.P : \mathtt{ap}_1 \to \mathtt{pp}_1}$$

where act does not bind free variable in $\mathtt{ap}_1 \to \mathtt{pp}_1$. From the inductive hypothesis, the statement follows for $\{X_1 : \mathtt{pp}_1, \ldots, X_i : \mathtt{pp}_i\} \vdash P : \mathtt{ap}_1 \to \mathtt{pp}_1$.

Notice that, if $P : \mathtt{ap}_1 \to \mathtt{pp}_1$ then there exist act_1 and P_1 such that: $P \stackrel{act_1}{\underset{\mathtt{fv(ap} \to \mathtt{pp)}}{\Longrightarrow}} P_1$ and $act_1 : \mathtt{pp}_1$ and $P_1 : \mathtt{pp}_1$. Moreover, for each act that does not bind variables in $\mathtt{fv(ap} \to \mathtt{pp)}, act.P \stackrel{act_1}{\underset{\mathtt{fv(ap} \to \mathtt{pp)}}{\Longrightarrow}} P_1$, hence $P : \mathtt{ap} \to \mathtt{pp} \Rightarrow act.P : \mathtt{ap} \to \mathtt{pp}$. Since the set variables that are bound by act do not change under every substitution, then the statement follows.

($\mathtt{RAct}_2$). Also that case follows by composing an inductive hypothesis and definition of $\mathbb{P}[\![\ ]\!]$.

(RCall). In that case the initial step of the actual proof is of the form:

$$\frac{X_1 : \mathtt{pp}_1, \ldots, X_i : \mathtt{pp}_n \vdash \overline{P}[\widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}] : \mathtt{pp} \quad \Delta \vdash P^1 : \mathtt{pp}_1 \ \ldots \ \Delta \vdash P^i : \mathtt{pp}_n}{\Delta \vdash A\langle P^1, \ldots, P^i, \widetilde{\ell}, \widetilde{v}\rangle : \mathtt{pp}}$$

where $\Delta = X'_1 : \mathtt{pp}'_1, \ldots, X'_k : \mathtt{pp}'_k$ and $A(X_1, \ldots, X_i, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} \overline{P}$. From the inductive hypothesis we have that:

—for all $P_1 : \mathtt{pp}_1, \ldots, P_i : \mathtt{pp}_i$ then $\overline{P}[\widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}][P_1/X_1, \ldots, P_i/X_i] : \mathtt{pp}$

—for all $P_1 : \mathtt{pp}'_1, \ldots, P_k : \mathtt{pp}'_k$ then for every $P^1[P_1/X'_1, \ldots, P_k/X'_k] : \mathtt{pp}_1, \ldots, P^i[P_1/X'_1, \ldots, P_k/X'_k] : \mathtt{pp}_i$.

Using these result we have that for every $P_1 : \mathtt{pp}'_1, \ldots, P_k : \mathtt{pp}'_k$:

$$\overline{P}[\widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}][P^1[P'_1/X'_1, \ldots, P'_k/X'_k], \ldots, P^n[P'_1/X'_1, \ldots, P'_k/X'_k]] : \mathtt{pp}$$

then

$$A\langle\widetilde{\ell}, P^1[P'_1/X'_1, \ldots, P'_k/X'_k], \ldots, P^n[P'_1/X'_1, \ldots, P'_k/X'_k], \widetilde{v}\rangle : \mathtt{pp}$$

and

$$A\langle\widetilde{\ell}, P^1, \ldots, P^n, \widetilde{v}\rangle[P'_1/X'_1, \ldots, P'_k/X'_k] : \mathtt{pp}$$ □

LEMMA 3.7. *If*

$$\begin{array}{l} \vdash P[P_1/X_1, \ldots P_n/X_n] : pp \\ \vdash act[P_1/X_1, \ldots P_n/X_n] : ap \\ \vdash t[P_1/X_1, \ldots P_n/X_n] : tp \end{array}$$

are provable, and $pp \neq X$, *then there exist* $pp_1, \ldots, pp_n$ *such that:*

—*for each* i, $P_i : pp_i$;

—*for each* i, $pp_i \in (\downarrow pp)^C$ *(respectively* $pp_i \in (\downarrow ap)^C$ *and* $pp_i \in (\downarrow tp)^C$*)*

—*the sequents*

$$X_1 : pp_1, \ldots, X_n : pp_n \vdash P : pp$$
$$X_1 : pp_1, \ldots, X_n : pp_n \vdash act : ap$$
$$X_1 : pp_1, \ldots, X_n : pp_n \vdash t : tp$$

are provable.

PROOF. We proceed by induction on the length of the proofs.

Base of Induction. If $\vdash P[P_1/X_1, \ldots P_n/X_n] : \mathtt{pp}$ is provable in 1 step then $\mathtt{pp} = \mathtt{1_P}$. Moreover, for each i, we let $\mathtt{pp}_i = \mathtt{1_P}$ and, since $\mathtt{1_P} \in \downarrow\ \mathtt{pp}^C, X_1 : \mathtt{1_P}, \ldots X_n : \mathtt{1_P} \vdash P : \mathtt{1_P}$. Every proof for $\vdash act[P_1/X_1, \ldots P_n/X_n] : \mathtt{ap}$ has a length greatest than 1, while if $\vdash t[P_1/X_1, \ldots P_n/X_n] : \mathtt{tp}$ is provable in one step then either $t = P$ and $\mathtt{tp} = \mathtt{1_P}, t = \ell$ and $\mathtt{tp} = \mathtt{lp}, t = !id$ and $\mathtt{tp} = !id$ $(id \in \{u, x, X\})$, $t = v$ and $\mathtt{tp} = \mathtt{vp}$ or $\mathtt{tp} = \mathtt{1_t}$. The statement holds for each of these cases.

Inductive Hypothesis. If $\vdash P[P_1/X_1, \ldots P_n/X_n] : \mathtt{pp} \vdash act[P_1/X_1, \ldots P_n/X_n] : \mathtt{ap}$ and $\vdash t[P_1/X_1, \ldots P_n/X_n] : \mathtt{tp}$ are provable within m steps then there exist $\mathtt{pp}_1, \ldots, \mathtt{pp}_n$ such that:

—for each i, $P_i : \mathtt{pp}_i$;
—for each i, $\mathtt{pp}_i \in (\downarrow \mathtt{pp})^C$, $\mathtt{pp}_i \in (\downarrow \mathtt{ap})^C$ and $\mathtt{pp}_i \in (\downarrow \mathtt{tp})^C$;
—the sequents

$$X_1 : \mathtt{pp}_1, \ldots, X_n : \mathtt{pp}_n \vdash P : \mathtt{pp}$$
$$X_1 : \mathtt{pp}_1, \ldots, X_n : \mathtt{pp}_n \vdash act : \mathtt{ap}$$
$$X_1 : \mathtt{pp}_1, \ldots, X_n : \mathtt{pp}_n \vdash t : \mathtt{tp}$$

are provable.

Inductive Step. The statement follows easily from the inductive hypothesis for act and t_1, t_2. The only interesting cases are those concerning processes. Let $\vdash P[P_1/X_1, \ldots P_n/X_n] : \mathtt{pp}$ be provable in $m + 1$ steps. If $\mathtt{pp} = \mathtt{pp}_1 \wedge \mathtt{pp}_2$ then the statement follows easily by using the inductive hypothesis. Let $\mathtt{pp} = \mathtt{ap} \to \mathtt{pp}_1$. We can distinguish three cases according to the last applied rule:

(R|,R+). $P = P^1|P^2$ or $P = P^1 + P^2$ then the proof has to start as follows:

$$\frac{\vdash P^i[P_1/X_1, \ldots P_n/X_n] : \mathtt{ap} \to \mathtt{pp}_1}{\vdash P[P_1/X_1, \ldots P_n/X_n] : \mathtt{ap} \to \mathtt{pp}_1}$$

by using the inductive hypothesis we have that there exists a proof for $X_1 : \mathtt{pp}_1 \ldots X_n : \mathtt{pp}_n \vdash P^i : \mathtt{pp}$. Hence there exists a proof for $X_1 : \mathtt{pp}_1 \ldots X_n : \mathtt{pp}_n \vdash P : \mathtt{pp}$.

($\mathtt{RAct}_1$). $P = act.P^i$ and $\vdash P^i[P_1/X_1, \ldots P_n/X_n] : \mathtt{ap} \to \mathtt{pp}$ is provable within n steps. In that case we can directly apply the inductive hypothesis.

($\mathtt{RAct}_2$). $P = act.P^i$ and $\vdash act[P_1/X_1, \ldots P_n/X_n] : \mathtt{ap}', \mathtt{ap}' =_\alpha \mathtt{ap}$ and $\vdash P^i[P_1/X_1, \ldots P_n/X_n] : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]$ are provable within n steps. By applying the

inductive hypothesis we have that there exists $\mathtt{pp}_1^1, \ldots, \mathtt{pp}_n^1$ and $\mathtt{pp}_1^2, \ldots, \mathtt{pp}_n^2$ such that

$$X_1 : \mathtt{pp}_1^1, \ldots, X_n : \mathtt{pp}_n^1 \vdash act : \mathtt{ap}'$$
$$X_1 : \mathtt{pp}_1^2, \ldots, X_n : \mathtt{pp}_n^2 \vdash P^i : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]$$

Please, notice that, for every i, $\mathtt{pp}_i^1 \wedge \mathtt{pp}_i^2 \in (\downarrow \mathtt{ap}' \rightarrow \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}])^C$. The proof, that we are looking for, starts as follows:

$$\dfrac{\dfrac{\dfrac{X_1 : \mathtt{pp}_1^1, \ldots \vdash act : \mathtt{ap}'}{\vdots}}{X_1 : \mathtt{pp}_1^1 \wedge \mathtt{pp}_1^2, \ldots \vdash act : \mathtt{ap}'} \quad \dfrac{\dfrac{X_1 : \mathtt{pp}_1^2, \ldots \vdash P^i : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]}{\vdots}}{X_1 : \mathtt{pp}_1^1 \wedge \mathtt{pp}_1^2, \ldots \vdash P^i : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]}}{X_1 : \mathtt{pp}_1^1 \wedge \mathtt{pp}_1^2, \ldots, X_n : \mathtt{pp}_n^1 \wedge \mathtt{pp}_n^2 \vdash P^i : \mathtt{ap} \rightarrow \mathtt{pp}}$$

(RCall). $P = A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{v} \rangle$ then the proof starts as follows:

$$\dfrac{\begin{array}{c} X^1 : \mathtt{pp}^1, \ldots, X^k : \mathtt{pp}^k \vdash P'[\widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}] : \mathtt{ap} \rightarrow \mathtt{pp}_1 \\ \vdash P^1[P_1/X_1, \ldots P_n/X_n] : \mathtt{pp}^1 \ldots \vdash P^k[P_1/X_1, \ldots P_n/X_n] : \mathtt{pp}^k \end{array}}{\vdash A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{v} \rangle [P_1/X_1, \ldots P_n/X_n] : \mathtt{ap} \rightarrow \mathtt{pp}_1}$$

where $\widetilde{P} = P^1, \ldots, P^k, A(X^1, \ldots, X^k, \widetilde{u}, \widetilde{x}) = P'$ and $\mathtt{pp}^1, \ldots, \mathtt{pp}^k \in (\downarrow \mathtt{pp})^C$. By applying the inductive hypothesis we have that there exists $\mathtt{pp}_1^1, \ldots, \mathtt{pp}_n^1, \ldots, \mathtt{pp}_1^k, \ldots, \mathtt{pp}_n^k$,such that $X_1 : \mathtt{pp}_1^1, \ldots, X_n : \mathtt{pp}_n^1 \vdash P^1 : \mathtt{pp}^1, \ldots, X_1 : \mathtt{pp}_1^k, \ldots, X_n : \mathtt{pp}_n^k \vdash P^k : \mathtt{pp}^k$ are provable, moreover $\mathtt{pp}_i^j \in (\downarrow \mathtt{pp}^j)^C \subseteq (\downarrow \mathtt{pp})^C$. Please notice that for every j $\mathtt{pp}_1^j \wedge \ldots \wedge \mathtt{pp}_n^j$ belongs to $(\downarrow \mathtt{pp})^C$. Moreover, similarly to the previous case, we have that

$$X_1 : \mathtt{pp}_1^1 \wedge \ldots \wedge \mathtt{pp}_1^k, \ldots X_n : \mathtt{pp}_n^1 \wedge \ldots \wedge \mathtt{pp}_n^k \vdash A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{v} \rangle : \mathtt{pp}$$

is provable. □

THEOREM 3.8 (COMPLETENESS). *For every P, act, t such that for every $P_1 : pp_1, \ldots, P_n : pp_n$ implies $P[P_1/X_1, \ldots, P_n/X_n] : pp, act[P_1/X_1, P_n/X_n] : ap$ $t[P_1/X_1, P_n/X_n] : tp$ then: $X_1 : pp_1, \ldots, X_n : pp_n \vdash P : pp, X_1 : pp_1, \ldots, X_n : pp_n \vdash act : ap$ and $X_1 : pp_1, \ldots, X_n : pp_n \vdash t : tp$ are provable.*

PROOF. We proceed by induction on the structure of pp, ap and tp. The only interesting case is $P : \mathtt{ap} \rightarrow \mathtt{pp}$, where we assume that for every $\mathtt{ap}'$ and $\mathtt{pp}'$ (subterms of $\mathtt{ap} \rightarrow \mathtt{pp}$) if $act' : \mathtt{ap}'$ and $P' : \mathtt{pp}'$ then $\vdash act' : \mathtt{ap}'$ and $\vdash P' : \mathtt{pp}'$ are provable.

If $P : \mathtt{ap} \rightarrow \mathtt{pp}$ then there exist $P_1, \ldots, P_n$ such that:

—$P = P_1$;
—$P_i \rightarrow_{\mathtt{fv}(\mathtt{ap} \rightarrow \mathtt{pp})} P_{i+1}$;
—$P_n = act'.P'$ and $act' : \mathtt{ap}'$, $\mathtt{ap}' =_\alpha \mathtt{ap}$ and $P' : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]$.

We prove that if $P : \mathtt{ap} \rightarrow \mathtt{pp}$ then there exists a proof for $\vdash P : \mathtt{ap} \rightarrow \mathtt{pp}$ by induction on n.

If $n = 1$ then $P = act'.P', act' : \mathtt{ap}', \mathtt{ap}' =_\alpha \mathtt{ap}$ and $P' : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]$. We have assumed that if $act' : \mathtt{ap}'$ and $P' : \mathtt{pp}'$ then $\vdash act' : \mathtt{ap}'$ and $\vdash P' : \mathtt{pp}'$ are provable, hence $\vdash act.P : \mathtt{ap} \to \mathtt{pp}$ is provable.

We suppose that for every P such that there exist $P_1, \ldots, P_k$ such that:

—$P = P_1$;
—$P_i \to_{\mathtt{fv}(\mathtt{ap} \to \mathtt{pp})} P_{i+1}$;
—$P_k = act'.P'$ and $act' : \mathtt{ap}', \mathtt{ap}' =_\alpha \mathtt{ap}$ and $P' : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]$.

then there exists a proof for $\vdash P : \mathtt{ap} \to \mathtt{pp}$.

Let P be such that $P_1, \ldots, P_k, P_{k+1}$ such that:

—$P = P_1$;
—$P_i \to_{\mathtt{fv}(\mathtt{ap} \to \mathtt{pp})} P_{i+1}$;
—$P_{k+1} = act'.P'$ and $act' : \mathtt{ap}', \mathtt{ap}' =_\alpha \mathtt{ap}$ and $P' : \mathtt{pp}[\mathtt{ap}'/\mathtt{ap}]$.

If $P = P^1|P^2$, $P = P^1 + P^2$ or $P = act_1.P^1$ the thesis follows directly from the inductive hypothesis. While if $P = A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{v}\rangle$ and $A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} P'$ then $P_2 = P'[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}]$. For the inductive hypothesis we have that there exists a proof for $\vdash P'[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}] : \mathtt{ap} \to \mathtt{pp}$. Then, by applying the previous lemma, we have that $X_1 : \mathtt{pp}_1, \ldots, X_n : \mathtt{pp}_n \vdash P'[\widetilde{\ell}/\widetilde{u}, \widetilde{v}/\widetilde{x}]$ is provable. Finally, let $\widetilde{P} = P_1, \ldots, P_n, \mathtt{pp}_1, \ldots, \mathtt{pp}_n \in \downarrow \mathtt{pp}$ and by assumption we have that $\vdash P_1 : \mathtt{pp}_1, \ldots, \vdash P_n : \mathtt{pp}_n$ are provable. □

We have proved that, for each P and $\mathtt{pp}, \vdash P : \mathtt{pp}$ is provable if and only if $P \in \mathbb{P}[\![\mathtt{pp}]\!]$. However, we also need to prove that we are able to decide if there exists a proof for $\vdash P : \mathtt{pp}$.

To obtain this result we first prove that the set of sequents that can be *reached* in a derivation is finite. Then we prove that if there exists a proof for a sequent with *loops* then a proof without loops can be found for the same sequent.

By composing these results we can prove decidability for the proof system in Table XIII. Indeed, search a proof for $\Delta \vdash P : \mathtt{pp}$ is equivalent to search paths, in the finite graph induced by the proof system, from $\Delta \vdash P : \mathtt{pp}$ to *axioms*.

LEMMA 3.9. *The set of sequents that are reachable in each deduction from* $\Delta \vdash P : pp$ *is finite.*

PROOF. Let $X_1 : \mathtt{pp}_1, \ldots, X_n : \mathtt{pp}_n \vdash P' : \mathtt{pp}'$ be a sequent reachable in a deduction from $\Delta \vdash P : \mathtt{pp}$ then:

—either $\mathtt{pp}'$ belongs to $(\downarrow \mathtt{pp})^C$ or there exists $\mathtt{pp}" \in (\downarrow \mathtt{pp})^C$ and $\mathtt{pp}'$ is obtained from $\mathtt{pp}"$ by replacing some variables of $\mathtt{pp}"$ with those that appear in P. This follows directly from the definition of proof system and from the definition of $(\downarrow \mathtt{pp})^C$;
—either $\mathtt{pp}_1, \ldots, \mathtt{pp}_n \in \downarrow \mathtt{pp}$ or $\mathtt{pp}_1, \ldots, \mathtt{pp}_n$ appear in Δ;
—P' is a subterm of P.

Since the set of variables, the set of localities, and the set of values that appear in P are finite, then the set of processes and process predicates that appear in a

deduction from $\Delta \vdash P : \mathtt{pp}$ are also finite (for every $\mathtt{pp}$ the sets $\downarrow \mathtt{pp}$ and $(\downarrow \mathtt{pp})^C$ are finite). Then the set of sequents that appear in a deduction from $\Delta \vdash P : \mathtt{pp}$ are finite too. □

LEMMA 3.10. *If $\Delta \vdash P : pp$ is provable then there exists a proof without loops.*

PROOF. If there exists a proof for $\Delta \vdash P : \mathtt{pp}$ with a loop then the proof would certainly have the following structure:

$$\dfrac{\dfrac{\dfrac{\dfrac{[C]}{\Delta_1 \vdash P_1 : \mathtt{pp}_1}}{\vdots}}{\Delta_1 \vdash P_1 : \mathtt{pp}_1} \quad [B]}{\dfrac{\ddots \qquad [A]}{\Delta \vdash P : \mathtt{pp}}}$$

This proof can be restructured as follows:

$$\dfrac{\dfrac{\dfrac{[C]}{\Delta_1 \vdash P_1 : \mathtt{pp}_1} \quad [B]}{\ddots \qquad [A]}}{\Delta \vdash P : \mathtt{pp}}$$

Applying the procedure iteratively we obtain a proof without loops. □

THEOREM 3.11. *The existence of a proof for $\Delta \vdash P : pp$, $\Delta \vdash act : ap$ and $\Delta \vdash t : tp$ is decidable.*

PROOF. The theorem follows easily from the two previous lemmata. Indeed, Lemma 3.9 establishes that the set of sequents that appears in a proof is finite. Then, the set of proofs without loops is finite too. Moreover, Lemma 3.10 states that if there is a proof for a sequent then there is also a proof without loops. Hence, searching a proof for $\Delta \vdash P : \mathtt{pp}$ corresponds to searching a valid proof in the finite set of proofs without loops. □

Let us consider process $A\langle\rangle$ where

$$A() \stackrel{def}{=} \mathbf{in}(!u_2)@s_1.\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.A\langle\rangle.$$

This process does not satisfy the predicate: $\mathtt{pp} = \mathtt{i}(!u)@s_1 \rightarrow \mathtt{e}(1_\mathtt{P})@u \rightarrow 1_\mathtt{P}$.

Indeed, if one looks for proofs without loops for $\vdash A\langle\rangle : \mathtt{pp}$ finds the following incomplete proofs:

$$\dfrac{\dfrac{\vdash !u_2 : !u_2 \ \vdash s_1 : s_1}{\vdash \mathbf{in}(!u_2)@s_1 : \mathtt{i}(!u_2)@s_1}\,\mathtt{RIn} \qquad \vdash \mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.A\langle\rangle : \mathtt{e}(1_\mathtt{P})@u_2 \rightarrow 1_\mathtt{P}(1)}{\vdash \mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.A\langle\rangle : \mathtt{i}(!u)@s_1 \rightarrow \mathtt{e}(1_\mathtt{P})@u \rightarrow 1_\mathtt{P}}\,\mathtt{RAct}_2$$

$$\dfrac{\vdash \mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.A\langle\rangle : \mathtt{i}(!u)@s_1 \rightarrow \mathtt{e}(1_\mathtt{P})@u \rightarrow 1_\mathtt{P}}{\vdash A\langle\rangle : \mathtt{i}(!u)@s_1 \rightarrow \mathtt{e}(1_\mathtt{P})@u \rightarrow 1_\mathtt{P}}\,\mathtt{RCall}$$

Table XIV. The Proof System

$$R1\quad \frac{\vdash N : \neg\phi_1 \qquad \vdash N : \neg\phi_2}{\vdash N : \neg(\phi_1 \vee \phi_2)} \qquad R2\quad \frac{\vdash N : \phi_i}{\vdash N : \phi_1 \vee \phi_2}$$

$$R3\quad \frac{\vdash N' : \phi\{\delta\}}{\vdash N : \langle\mathcal{A}\rangle\phi}\ N \succ\!\!\xrightarrow{a} N', (a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \qquad R4\quad \frac{\vdash N : \phi}{\vdash N : \neg\neg\phi}$$

$$R5\quad \frac{\vdash N_1 : \neg\phi\{\delta_1\} \quad \vdash N_2 : \neg\phi\{\delta_2\} \quad \dots}{\vdash N : \neg\langle\mathcal{A}\rangle\phi}\ \left[\forall(a_i;\delta_i) \in \mathbb{A}[\![\mathcal{A}]\!] : \forall N_i \in \left\{ N' | N \succ\!\!\xrightarrow{a_i} N' \right\}\right]$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash A\langle\rangle : \mathtt{i}(!u)@s_1 \to \mathtt{e}(1_\mathtt{P})@u \to 1_\mathtt{P}(2)}{\vdash \mathbf{eval}(P)@u_2.A\langle\rangle : \mathtt{i}(!u)@s_1 \to \mathtt{e}(1_\mathtt{P})@u \to 1_\mathtt{P}}\,\mathtt{RAct}_1}{\vdash \mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.A\langle\rangle : \mathtt{i}(!u)@s_1 \to \mathtt{e}(1_\mathtt{P})@u \to 1_\mathtt{P}}\,\mathtt{RAct}_1}{\vdash \mathbf{in}(!u_2)@s_1.\mathbf{read}(!u_2)@s_2.\mathbf{eval}(P)@u_2.A\langle\rangle : \mathtt{i}(!u)@s_1 \to \mathtt{e}(1_\mathtt{P})@u \to 1_\mathtt{P}}\,\mathtt{RAct}_1}{\vdash A\langle\rangle : \mathtt{i}(!u)@s_1 \to \mathtt{e}(1_\mathtt{P})@u \to 1_\mathtt{P}}\,\mathtt{RCall}$$

where neither proofs can be completed. In the first one, indeed, for the sequent (1), rule $\mathtt{RAct}_1$ cannot be applied, indeed $\mathbf{read}(!u_2)@s_2$ binds variable u_2, which is free in $\mathtt{e}(1_\mathtt{P})@u_2$, while applying rule $\mathtt{RAct}_2$ yields the sequent $\vdash \mathbf{read}(!u_2)@s_2 : \mathtt{e}(1_\mathtt{P})@u_2$, which is not an axiom and for which there is no rule to apply.

In the second proof, instead, the sequent (2) is the same at the beginning of the proof. Hence, each proof obtained from (2) will contain a loop.

Finally, there are no correct proofs, without loops, for $\vdash A\langle\rangle : \mathtt{i}(!u)@s_1 \to \mathtt{e}(1_\mathtt{P})@u \to 1_\mathtt{P}$. Hence, $\vdash A\langle\rangle$ does not satisfy $\mathtt{i}(!u)@s_1 \to \mathtt{e}(1_\mathtt{P})@u \to 1_\mathtt{P}$.

3.3 The Proof System for $\mathcal{L}$

We now introduce a proof system for the finite fragment of the logic. The proof rules operate on *sequents* of the form $\vdash N : \phi$. We shall use π to refer to sequents and Π to refer to proofs.

The proof system is presented in Table XIV. Rules reflect the intuitive semantics of logical operators. Rule $R1$ states that if one proves that N satisfies both $\neg\phi_1$ and $\neg\phi_2$ then N satisfies $\neg(\phi_1 \vee \phi_2)$. Rule $R2$, instead, states that if N satisfies ϕ_1 or ϕ_2 then N satisfies $\phi_1 \vee \phi_2$. The idempotence of $\neg$ is established by rule $R4$. Finally, rules $R3$ and $R5$ are those that regard modal operator $\langle\cdot\rangle$.

Definition 3.12.

(1) A sequent $\vdash N : \phi$ is *successful* if
   —$\phi = \mathbf{tt}$;
   —or $\phi = \neg\langle\mathcal{A}\rangle\phi'$, and $\forall(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \nexists N'.N \succ\!\!\xrightarrow{a} N'$;
   —or $\phi = \mathtt{tp}@s$ and there exists et such that $N \xrightarrow{et@s}\!\!\!\!\succ N'$ and $et : \mathtt{tp}$;
   —or $\phi = \neg\,\mathtt{tp}@s$ and for all et such that $N \xrightarrow{et@s}\!\!\!\!\succ N'$ then $\neg(et : \mathtt{tp})$.

(2) Π is a successful proof for π if the following conditions hold:
—Π is built using the rules on Table XIV;
—and π is the root of Π;
—and every leaf on Π is a successful sequent.

3.4 Soundness and Completeness

In this section, we prove that for every closed formula ϕ and for every net N if there exists a valid proof for $\vdash N : \phi$ then $N \models \phi$; moreover if $N \models \phi$ then there exists a valid proof for $\vdash N : \phi$.

3.4.1 *Soundness*

THEOREM 3.13. *Let ϕ be a formula. If there exists a proof Π for $\vdash N : \phi$ then $N \in \mathbb{M}[\![\phi]\!]$.*

PROOF. The proof goes by induction on the length of Π.

Base of Induction. If length of Π is 0 then it consists of one successful sequent $\vdash N : \phi$. We can have:

—$\phi = \mathbf{tt}$;
—or $\phi = \mathtt{tp}@s$ and there exist et such that $N \xrightarrow{et@s} N'$ and $et : \mathtt{tp}$;
—or $\phi = \neg\, \mathtt{tp}@s$ and for all et such that $N \xrightarrow{et@s} N'$ then $\neg(et : \mathtt{tp})$.
—or $\phi = \neg\langle\mathcal{A}\rangle\phi'$ and $\forall(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!]\ \nexists N' : N \xrightarrow{a} N'$.

For each of these cases we have that $N \in \mathbb{M}[\![\phi]\!]$.

Inductive Hypothesis. For every net N and formula ϕ, if $\vdash N : \phi$ is provable in less then n steps then $N \in \mathbb{M}[\![\phi]\!]$.

Inductive Step. Let $\vdash N : \phi$ be provable in $n+1$ steps. We have to consider different cases depending upon the last applied rule.

R1. Then $\phi = \neg(\phi_1 \vee \phi_2)$ and the proof starts as follows:

$$\frac{\vdash N : \neg\phi_1 \qquad \vdash N : \neg\phi_2}{\vdash N : \neg(\phi_1 \vee \phi_2)}$$

By using the inductive hypothesis we have that $N \in \mathbb{M}[\![\neg\phi_1]\!]$ and $N \in \mathbb{M}[\![\neg\phi_2]\!]$. Then:

$$\begin{aligned} N &\in (Net - \mathbb{M}[\![\phi_1]\!]) \cap (Net - \mathbb{M}[\![\phi_2]\!]) \\ &= Net - (\mathbb{M}[\![\phi_1]\!] \cup \mathbb{M}[\![\phi_2]\!]) \\ &= Net - (\mathbb{M}[\![\phi_1 \vee \phi_2]\!]) \\ &= \mathbb{M}[\![\neg(\phi_1 \vee \phi_2)]\!] \end{aligned}$$

R2. Then $\phi = \phi_1 \vee \phi_2$ and the proof starts as follows:

$$\frac{\vdash N : \phi_i}{\vdash N : \phi_1 \vee \phi_2}$$

By using the inductive hypothesis we have that $N \in \mathbb{M}[\![\phi_1]\!]$ moreover we have that:

$$\begin{aligned} N &\in \mathbb{M}[\![\phi_1]\!] \\ &\subseteq \mathbb{M}[\![\phi_1]\!] \cup \mathbb{M}[\![\phi_2]\!] \\ &= \mathbb{M}[\![\phi_1 \vee \phi_2]\!] \end{aligned}$$

R3. Then $\phi = \langle \mathcal{A} \rangle \phi_1$ and the proof is such that:

$$\frac{\vdash N' : \phi_1\{\delta\}}{\vdash N : \langle \mathcal{A} \rangle \phi_1}$$

where there exists $(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!]$ and N' for which $N \stackrel{a}{\succ\!\!\longrightarrow} N'$. By using the inductive hypothesis we have that $N' \in \mathbb{M}[\![\phi_1\{\delta\}]\!]$. Moreover by using definition of $\mathbb{M}[\![\ ]\!]$ we have that $N \in \mathbb{M}[\![\langle \mathcal{A} \rangle \phi_1]\!]$.

R4. Then $\phi = \neg\neg\phi_1$ and the proof is such that:

$$\frac{\vdash N : \phi_1}{\vdash N : \neg\neg\phi_1}$$

We have that $N \in \mathbb{M}[\![\phi_1]\!]$ and $\mathbb{M}[\![\neg\neg\phi_1]\!] = \mathbb{M}[\![\phi_1]\!]$.

R5. Then $\phi = \neg\langle \mathcal{A} \rangle \phi'$ and the proof is such that:

$$\frac{\vdash N_1 : \neg\phi'\{\delta_1\} \quad \ldots \quad \vdash N_k : \neg\phi'\{\delta_k\}}{\vdash N : \neg\langle \mathcal{A} \rangle \phi}$$

where

$$\{N_1, \ldots, N_k\} = \bigcup_{(a_i;\delta_i) \in \mathbb{A}[\![\mathcal{A}]\!]} \{N' | N \stackrel{a_i}{\succ\!\!\longrightarrow} N'\}$$

Using the inductive hypothesis we have that

$$\forall (a_i;\delta_i) \in \mathbb{A}[\![\mathcal{A}]\!], \forall N_i \in \{N' | N \stackrel{a_i}{\succ\!\!\longrightarrow} N'\} : N_i \in \mathbb{M}[\![\neg\phi'\{\delta_i\}]\!]$$

and, for Theorem 2.3, $\{N' | N \stackrel{a}{\succ\!\!\longrightarrow} N'\}$ is finite. Equivalently

$$\neg(\exists (a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \exists N' \in \{N' | N \stackrel{a}{\succ\!\!\longrightarrow} N'\} : N' \in \mathbb{M}[\![\phi'\{\delta\}]\!])$$

then $N \notin \mathbb{M}[\![\langle \mathcal{A} \rangle \phi']\!] \Rightarrow N \in \mathbb{M}[\![\neg\langle \mathcal{A} \rangle \phi']\!]$. □

3.4.2 *Completeness*

LEMMA 3.14. *Let N be a net, then one between $\vdash N : \phi$ and $\vdash N : \neg\phi$ is provable.*

PROOF. The lemma is provable by induction on the syntax of ϕ.

Base of Induction. Let $\phi = \mathtt{tp}@s$ then either $\vdash N : \mathtt{tp}@s$ or $\vdash N : \neg\,\mathtt{tp}@s$ indeed either $N \stackrel{et@s}{\longrightarrow\!\!\!\!\!\rightarrow} N'$ and $et : \mathtt{tp}$ or for all et such that $N \stackrel{et@s}{\longrightarrow\!\!\!\!\!\rightarrow} N'$ then such that $\neg(et : \mathtt{tp})$.

Inductive Hypothesis. Let ϕ_1 and ϕ_2 be such that either $\vdash N : \phi_i$ or $\vdash N : \neg\phi_i$ are provable $(i = 1, 2)$.

Inductive Step.

—if $\phi = \phi_1 \vee \phi_2$ then $\vdash N : \phi$ is provable if and only if $\vdash N : \phi_1$ or $\vdash N : \phi_2$ are provable. For the inductive hypothesis either $\vdash N : \phi_i$ or $\vdash N : \neg\phi_i$ are provable. Then if $\vdash N : \phi_1 \vee \phi_2$ is provable then it cannot be that both $\vdash N : \neg\phi_1$ and $\vdash N : \neg\phi_2$ are provable, then $\vdash N : \neg(\phi_1 \vee \phi_2)$ is not provable. On the other hand, if $\vdash N : \neg(\phi_1 \vee \phi_2)$ is provable, then both $\vdash N : \neg\phi_1$ and $\vdash N : \neg\phi_2$ are provable. From the inductive hypothesis we have that both $\vdash N : \phi_1$ and $\vdash N : \phi_2$ are not provable and $\vdash N : \phi_1 \vee \phi_2$ is not provable.

—if $\phi = \langle\mathcal{A}\rangle\phi_1$ and $\vdash N : \langle\mathcal{A}\rangle\phi_1$ is provable if there exists N' such that $N \succ \xrightarrow{a} N', (a;\delta) \in \mathbb{M}[\![\mathcal{A}]\!]$ and $\vdash N'\phi_1\{\delta\}$. By using the inductive hypothesis we have that $\vdash N' : \neg\phi_1\{\delta\}$ is not provable hence $\vdash N : \neg\langle\mathcal{A}\rangle\phi_1$ is also not provable. Conversely if $\vdash N : \langle\mathcal{A}\rangle\phi_1$ is not provable then for every N' such that $N \succ\!\!\rightarrow^{a} N'$ and $(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \vdash N' : \phi_1\{\delta\}$ is not provable. By using the inductive hypothesis we have that, for every $N', \vdash N' : \neg\phi_1\{\delta\}$ is provable. We have that the set of N' such that $N \succ \xrightarrow{a} N'$ and $(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \vdash N' : \phi_1\{\delta\}$ is finite, then $\vdash N' : \neg\phi$ is provable.

—$\phi = \neg\phi'$, the thesis follows directly from previous cases and by using $\neg\neg\phi = \phi$. □

THEOREM 3.15. *Let N be a net and let ϕ be a closed formula. If $N \in \mathbb{M}[\![\phi]\!]$ then $\vdash N : \phi$ is provable.*

PROOF. We suppose that $N \in \mathbb{M}[\![\phi]\!]$ and that $\vdash N : \phi$ is not provable. For the previous lemma we have that $\vdash N : \neg\phi$ is provable, and from the soundness we have also that $N \in \mathbb{M}[\![\neg\phi]\!]$. Thus from the definition of $\mathbb{M}[\![\cdot]\!]$ we have that $N \in Net - \mathbb{M}[\![\phi]\!]$, which is in contradiction with the hypothesis. □

4. RECURSIVE FORMULAE

In this section, we extend the logic with an operator for recursion. We define $\mathcal{L}^*$ as the set of formulae generable from the following grammar, where κ belongs to the set of logical variables $VLog$.

$$\phi \ ::= \ \mathbf{tt} \mid \mathtt{tp}@\sigma \mid \langle\mathcal{A}\rangle\phi \mid \kappa \mid \nu\kappa.\phi \mid \phi \vee \phi \mid \neg\phi$$

To guarantee well definedness of the interpretation function of formulae, we shall assume that no variable κ occurs negatively (i.e. under the scope of an odd number of $\neg$ operators) in ϕ.

4.1 Semantics

We have to extend the interpretation function $\mathbb{M}[\![]\!]$ with two new parameters: a substitution environment and a logical environment. This interpretation function follows the schema presented in Rathke and Hennessy [1997].

Definition 4.1. We define the *logical environment* Env as

$$Env \subseteq [VLog \to Subst \to 2^{Net}]$$

We also use ϵ, sometimes with indexes, to denote elements of Env.

We define $\mathbb{M}[\![\cdot]\!] : \mathcal{L}^* \to Env \to Subst \to 2^{Net}$ to denote the set of nets that are models of a formula ϕ. The function $\mathbb{M}[\![\cdot]\!]$ is defined as follows:

—$\mathbb{M}[\![\mathbf{tt}]\!]\epsilon\delta = Net$;
—$\mathbb{M}[\![\kappa]\!]\epsilon\delta = \epsilon(\kappa)\delta$;
—$\mathbb{M}[\![\ \mathtt{tp}@\sigma\ ]\!]\epsilon\delta = \{N | s = \sigma\{\delta\}, \exists et.\ N \xrightarrow{et@s} N', et : \mathtt{tp}\{\delta\}\}$;
—$\mathbb{M}[\![\langle\mathcal{A}\rangle\phi]\!]\epsilon\delta = f_{\mathcal{A}\{\delta\}}(\mathbb{M}[\![\phi]\!]\epsilon)$ where for each $\mathcal{A}$,

$$f_{\mathcal{A}} : (Subst \to 2^{Net}) \to (Subst \to 2^{Net})$$

is defined as follows:

$$f_{\mathcal{A}}(g)\delta = \bigcup_{(a;\delta')\in\mathbb{A}[\![\mathcal{A}]\!]} \{N' | \exists N.\ N' \overset{a}{\rightarrowtail} N\ N \in g\delta' \cdot \delta\}$$

—$\mathbb{M}[\![\phi_1 \vee \phi_2]\!]\epsilon\delta = \mathbb{M}[\![\phi_1]\!]\epsilon\delta \cup \mathbb{M}[\![\phi_2]\!]\epsilon\delta$;
—$\mathbb{M}[\![\neg\phi]\!]\epsilon\delta = Net - \mathbb{M}[\![\phi]\!]\epsilon\delta$;
—$\mathbb{M}[\![\nu\kappa.\phi]\!]\epsilon\delta = \nu f^{\phi}_{\kappa,\epsilon}\delta$ where:

(1) $f^{\phi}_{\kappa,\epsilon} : [Subst \to 2^{Net}] \to [Subst \to 2^{Net}]$ is defined as follows:

$$f^{\phi}_{\kappa,\epsilon}(g) = \mathbb{M}[\![\phi]\!]\epsilon \cdot [\kappa \mapsto g]$$

(2) $\nu f^{\phi}_{\kappa,\epsilon} = \bigcup\{g | g \subseteq f^{\phi}_{\kappa,\epsilon}(g)\}$ where $g_1 \subseteq g_2$ if and only if for all δ $g_1(\delta) \subseteq g_2(\delta)$.

We will write $N \models \phi$ if $N \in \mathbb{M}[\![\phi]\!]\epsilon_0\delta_0$ where $\epsilon_0 = \lambda\kappa.\emptyset$ and $\delta_0 = []$.

Notice that introducing substitution like an explicit parameters in the definition of the interpretation function, and assuming that all logical variables appear in formulae positively, permits defining function $\mathbb{M}[\![]\!]$ like composition of monotone functions in $((Subst \to 2^{Net}) \to (Subst \to 2^{Net}), \subseteq)$, where $g_1 \subseteq g_2$ $(g_1, g_2 \in Subst \to 2^{Net})$ if for each δ $g_1(\delta) \subseteq g_2(\delta)$. Moreover, since $\langle(Subst \to 2^{Net}) \to (Subst \to 2^{Net}), \subseteq\rangle$ is a complete lattice, we can use Tarski's Fixed Point Theorem that guarantees existence of a unique *maximal* fixed point for $\mathbb{M}[\![\phi]\!]\epsilon$, whenever logical variables that appear in ϕ are all positive.

4.2 The Proof System

We now introduce a tableau-based proof system for the $\mathcal{L}^*$ formulae. A tableau based proof system for μ-calculus has been introduced in Cleaveland [1990]. Sequents that are used in tableau proofs maintain information about the systems involved in *recursive* properties. This permits avoiding proofs with *loops*.

Sequents presented in Section 3.3 are extended with a set of hypotheses H, that is a set of pairs $N : \phi$. Hence new rules operate on *sequents* of the form $H \vdash N : \phi$. The proof system is defined in Table XV. Rules **R1-R5** are essentially the same as in Table XIV, where we have the rules for the finitary logics. The only difference is the addition of the hypothesis H. Rules **R6** and **R7** are those introduced for handling recursion. There we use the *sub-term* relation ($\prec$) and the *structural congruence* ($\equiv$), which are defined below.

Definition 4.2. Let ϕ_1 and ϕ_2 be formulae; we say that ϕ_1 is an *immediate sub-term* of ϕ_2, written $\phi_1 \prec_I \phi_2$, if one of the following holds:

Table XV. The Proof System

$$R1 \quad \frac{H \vdash N : \neg\phi_1 \qquad H \vdash N : \neg\phi_2}{H \vdash N : \neg(\phi_1 \vee \phi_2)} \qquad R2 \quad \frac{H \vdash N : \phi_i}{H \vdash N : \phi_1 \vee \phi_2}$$

$$R3 \quad \frac{H \vdash N' : \phi\{\delta\}}{H \vdash N : \langle\mathcal{A}\rangle\phi} \; N \succ\!\!\xrightarrow{a} N', (a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \qquad R4 \quad \frac{H \vdash N : \phi}{H \vdash N : \neg\neg\phi}$$

$$R5 \quad \frac{H \vdash N_1 : \neg\phi\{\delta_1\} \quad H \vdash N_2 : \neg\phi\{\delta_2\} \quad \dots}{H \vdash N : \neg\langle\mathcal{A}\rangle\phi} \left[\forall(a_i;\delta_i) \in \mathbb{A}[\![\mathcal{A}]\!] : \forall N_i \in \left\{ N' | N \succ\!\!\xrightarrow{a_i} N' \right\}\right]$$

$$R6 \quad \frac{H' \cup \{N : \nu\kappa.\phi\} \vdash N : \neg\phi[\nu\kappa.\phi/\kappa]}{H \vdash N : \neg\nu\kappa.\phi} \; [N' \equiv N, N' : \nu\kappa.\phi \notin H]$$

$$R7 \quad \frac{H' \cup \{N' : \nu\kappa.\phi\} \vdash N : \phi[\nu\kappa.\phi/\kappa]}{H \vdash N : \nu\kappa.\phi} \; [N' \equiv N, N' : \nu\kappa.\phi \notin H]$$

where $H' = H - \{ N' : \phi' | \nu\kappa.\phi \prec \phi' \}$

(1) $\phi_2 = \neg\phi_1$;
(2) $\phi_2 = \phi_1 \vee \phi_3$ or $\phi = \phi_3 \vee \phi_1$, for some ϕ_3;
(3) $\phi_2 = \langle\mathcal{A}\rangle\phi_1$;
(4) $\phi_2 = \nu\kappa.\phi_1$.

We use $\prec$ for the transitive closure of $\prec_I$, and $\preceq$ for the transitive and reflexive closure of $\prec_I$.

Definition 4.3. We define $\equiv$ as the least congruence defined as follows:

(1) $N \equiv N$;
(2) $N_1 \parallel N_2 \equiv N_2 \parallel N_1$;
(3) $(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$;
(4) $s ::_\rho P_1 \parallel s ::_\rho P_2 \equiv s ::_\rho P_1 | P_2$;
(5) $s ::_\rho P | \mathbf{nil} \equiv s ::_\rho P$.

Let $\mathcal{N}$ be a set of KLAIM nets, we will sometimes use $\mathcal{N}/\equiv$ like the set $\mathcal{N}$ quotiented with respect to the equivalence $\equiv$.

As for formulae without recursion, we need to define successful sequents and proofs.

Definition 4.4.

(1) A sequent $H \vdash N : \phi$ is *successful* if
— $\phi = \mathbf{tt}$;
— or $\phi = \nu\kappa.\phi'$ and $\exists N'.N' \equiv N$ and $N' : \nu\kappa.\phi' \in H$;
— or $\phi = \neg\langle\mathcal{A}\rangle\phi'$, and $\forall(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \nexists N'.N \succ\!\!\xrightarrow{a} N'$;
— or $\phi =$ tp@s and there exists et such that $N \xrightarrow{et@s} N'$ and $et :$ tp;
— or $\phi = \neg$ tp@s and for all et such that $N \xrightarrow{et@s} N'$ then $\neg(et :$ tp).
(2) Π is a successful proof for π if the following conditions hold:
— Π is built using the rules in Table XV;

—and π is the root of Π;
—and every leaf on Π is a successful sequent.

4.3 Soundness and Completeness

The proof of soundness and completeness follows the scheme of Section 3.4. However, in order to prove recursive properties for nets we could have to consider an infinite number of sequents. What we shall prove is that for every formula ϕ:

—if there exists a successful proof for $\emptyset \vdash N : \phi$ then $N \models \phi$;
—if $N \models \phi$ and the set $\mathcal{N} = \{N' | N \succ\!\!\rightarrow^* N'\}/\equiv$ is finite then there exists a successful proof for $\emptyset \vdash N : \phi$;
—if $N \models \phi$ and ϕ satisfies some syntactic constraints then there exists a successful proof for $\emptyset \vdash N : \phi$.

First we have to extend the definition of $\mathbb{M}[\![\ ]\!]$ in order to consider the set of hypothesis H. We define $\mathbb{M}[\![\phi]\!]^H$ as follows:

—$\mathbb{M}[\![\mathbf{tt}]\!]^H \epsilon\delta = Net$;
—$\mathbb{M}[\![\kappa]\!]^H \epsilon\delta = e(\kappa)\delta$
—$\mathbb{M}[\![\ \mathtt{tp}@\sigma]\!]^H \epsilon\delta = \{N | s = \sigma\{\delta\}, \exists et.\ N \xrightarrow{et@s} N', et :: \mathtt{tp}\{\delta\}\}$;
—$\mathbb{M}[\![\langle\mathcal{A}\rangle\phi]\!]^H \epsilon\delta = \{N | \exists a, \delta', N' : N \succ\!\!\xrightarrow{a} N', (a;\delta') \in \mathbb{A}[\![\mathcal{A}\{\delta\}]\!], N' \in \mathbb{M}[\![\phi]\!]^H \epsilon\delta \cdot \delta'\}$;
—$\mathbb{M}[\![\phi_1 \vee \phi_2]\!]^H \epsilon\delta = \mathbb{M}[\![\phi_1]\!]^H \epsilon\delta \cup \mathbb{M}[\![\phi_2]\!]^H \epsilon\delta$;
—$\mathbb{M}[\![\neg\phi]\!]^H \epsilon\delta = Net - \mathbb{M}[\![\phi]\!]^H \epsilon\delta$;
—$\mathbb{M}[\![\nu\kappa.\phi]\!]^H \epsilon\delta = \nu f^{\phi,h}_{\kappa,\epsilon}\delta \cup h\delta$ where:
(1) $f^{\phi,h}_{\kappa,\epsilon} : [Subst \to 2^{Net}] \to [Subst \to 2^{Net}]$ is defined as follows:
$$f^{\phi,h}_{\kappa,\epsilon}(g) = f^{\phi}_{\kappa,\epsilon}(g \cup h)$$
(2) $h : Subst \to Nets^*$ is defined as follows:
$$h\delta = \{N | N : \nu\kappa.\phi\{\delta\} \in H\}$$
(3) $\nu f^{\phi,h}_{\kappa,\epsilon} = \bigcup\{g | g \subseteq f^{\phi,h}_{\kappa,\epsilon}(g)\}$.

Please notice that if $H = \emptyset$ then $\mathbb{M}[\![\phi]\!]^H \epsilon\delta = \mathbb{M}[\![\phi]\!]\epsilon\delta$.

We introduce two technical lemmata. The first guarantees that the set of hypotheses H does not alter the interpretation of the formulae. The second states that two structural equivalent nets are models for the same set of formulae.

LEMMA 4.5. *For $f^{\phi,h}_{\kappa,\epsilon}$ the following properties hold:*

—*$f^{\phi,h}_{\kappa,\epsilon}$ is continuous in $((Subst \to 2^{Net}) \to (Subst \to 2^{Net}), \subseteq)$;*
—*let $h = h_1 \cup h_2, h_1 \subseteq \nu f^{\phi,h}_{\kappa,\epsilon}$ if and only if $h_1 \subseteq \nu f^{\phi,h_2}_{\kappa,\epsilon}$;*
—*if $h_1 \subseteq \nu f^{\phi,h_2}_{\kappa,\epsilon}$ then $\nu f^{\phi,h_2}_{\kappa,\epsilon} = \nu f^{\phi,h}_{\kappa,\epsilon}$.*

PROOF. (1) This derives directly from the continuity of $f^{\phi}_{\kappa,\epsilon}$.
(2) ($\Rightarrow$) $h_1 \subseteq \nu f^{\phi,h}_{\kappa,\epsilon}$ if and only if there exists g such that:
$$h_1 \subseteq g \subseteq f^{\phi,h}_{\kappa,\epsilon}(g) = f^{\phi}_{\kappa,\epsilon}(g \cup h)$$

if $h_1 \subseteq g$ then $g \cup h = g \cup h_2$ and

$$g \subseteq f^{\phi}_{\kappa,\epsilon}(g \cup h_2) = f^{\phi,h_2}_{\kappa,\epsilon}(g)$$

then $h_1 \subseteq g \subseteq \nu f^{\phi,h_2}_{\kappa,\epsilon}$.
($\Leftarrow$) If $h_1 \subseteq \nu f^{\phi,h_2}_{\kappa,\epsilon}$ then there exists g such that

$$\begin{aligned} h_1 \subseteq g \subseteq f^{\phi,h_2}_{\kappa,\epsilon}(g) &= f^{\phi}_{\kappa,\epsilon}(g \cup h_2) \\ &= f^{\phi}_{\kappa,\epsilon}(g \cup h) \\ &= f^{\phi,h}_{\kappa,\epsilon}(g) \\ &\Rightarrow g \subseteq f^{\phi,h}_{\kappa,\epsilon}(g) \end{aligned}$$

(3) Let $h_1 \subseteq \nu f^{\phi,h_2}_{\kappa,\epsilon}$. We have that:

$$f^{\phi,h_2}_{\kappa,\epsilon}\left(\nu f^{\phi,h_2}_{\kappa,\epsilon}\right) = \nu f^{\phi,h_2}_{\kappa,\epsilon} \tag{1}$$

$$\nu f^{\phi,h_2}_{\kappa,\epsilon} = \nu f^{\phi,h_2}_{\kappa,\epsilon} \cup h_1 \tag{2}$$

then from 1 and 2 we have that

$$\nu f^{\phi,h_2}_{\kappa,\epsilon} = f^{\phi,h_2}_{\kappa,\epsilon}\left(\nu f^{\phi,h_2}_{\kappa,\epsilon} \cup h_1\right) = f^{\phi,h}_{\kappa,\epsilon}\left(\nu f^{\phi,h_2}_{\kappa,\epsilon}\right)$$

then $\nu f^{\phi,h}_{\kappa,\epsilon} \subseteq \nu f^{\phi,h_2}_{\kappa,\epsilon}$. On the other hand we have that

$$\begin{aligned} \nu f^{\phi,h}_{\kappa,\epsilon} &= f^{\phi,h}_{\kappa,\epsilon}\left(\nu f^{\phi,h}_{\kappa,\epsilon}\right) \\ &= f^{\phi}_{\kappa,\epsilon}\left(\nu f^{\phi,h}_{\kappa,\epsilon} \cup h\right) \\ &= f^{\phi}_{\kappa,\epsilon}\left(\nu f^{\phi,h}_{\kappa,\epsilon} \cup h_2\right) \\ &= f^{\phi,h_2}_{\kappa,\epsilon}\left(\nu f^{\phi,h}_{\kappa,\epsilon}\right) \end{aligned}$$

then $\nu f^{\phi,h_2}_{\kappa,\epsilon} \subseteq \nu f^{\phi,h}_{\kappa,\epsilon}$ and $\nu f^{\phi,h_2}_{\kappa,\epsilon} \subseteq \nu f^{\phi,h}_{\kappa,\epsilon}$. □

LEMMA 4.6. *Let N_1 and N_2 be such that $N_1 \equiv N_2$, then for every formula ϕ, $N_1 \in \mathbb{M}[\![\phi]\!] \Leftrightarrow N_2 \in \mathbb{M}[\![\phi]\!]$.*

PROOF. The proof follows directly by induction on the structure of ϕ and by observing that, since $\equiv$ does not involve processes and tuples argument of actions, if $N_1 \equiv N_2$ then $N_1 \overset{a}{\rightarrowtail} N_1' \Leftrightarrow N_2 \overset{a}{\rightarrowtail} N_2'$ and $N_1' \equiv N_2'$. □

4.3.1 *Soundness.* Soundness can be proved analogously to 3.4.1. However, in order to treat difficulties related to recursive functions we have to introduce the following lemma.

LEMMA 4.7. *For every formula ϕ, set of hypotheses H, and closed formula ϕ' such that no strict subformula of $\phi'\{\delta\}$ is in H:*

$$\mathbb{M}[\![\phi[\phi'/\kappa]]\!]^H \epsilon\delta = f^{\phi}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]^H \epsilon)\delta$$

PROOF. The lemma is provable by induction on the structure of the formula ϕ.

Base of Induction. If $\phi = \kappa$, $\phi =$ tp@s or $\phi = \neg$ tp@s then the thesis follows easily.

Inductive Hypothesis. Let ϕ_1 and ϕ_2 be such that

$$\mathbb{M}[\![\phi_i[\phi/\kappa]]\!]^H \epsilon\delta = f^{\phi_i}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]^H \epsilon)\delta$$

Inductive Step.

—$\phi = \phi_1 \vee \phi_2$:

$$\begin{aligned}\mathbb{M}[\![\phi[\phi'/\kappa]]\!]^H\epsilon\delta &= \mathbb{M}[\![\phi_1[\phi'/\kappa] \vee \phi_2[\phi'/\kappa]]\!]\epsilon\delta\\ &= \mathbb{M}[\![\phi_1[\phi'/\kappa]]\!]\epsilon\delta \cup \mathbb{M}[\![\phi_2[\phi'/\kappa]]\!]\epsilon\delta\\ &= f^{\phi_1}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]\epsilon)\delta \cup f^{\phi_2}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]\epsilon)\delta\\ &= f^{\phi_1\vee\phi_2}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]\epsilon)\delta\end{aligned}$$

—$\phi = \nu\kappa_1\phi_1$:

$$\mathbb{M}[\![\phi[\phi'/\kappa]]\!]^H\epsilon\delta = \nu f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon} \cup h\delta$$

where $h\delta = \{N | N : \nu\kappa_1.\phi_1[\phi'/\kappa]\{\delta\} \in H\}$. For the hypothesis no formula that involves $\phi'\{\delta\}$ as strict subformula is in H, then, $h\delta = \emptyset$, then

$$\begin{aligned}\nu f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon} \cup h\delta &= \nu f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon}\\ &= f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon}\left(\nu f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon}\right)\\ &= \mathbb{M}[\![\phi_1[\phi'/\kappa]]\!]^H e[\kappa_1 \mapsto \nu f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon}\\ &= \mathbb{M}[\![\phi_1]\!]^H\epsilon\left[\kappa_1 \mapsto \nu f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon}\right]\left[\kappa \mapsto \mathbb{M}[\![\phi']\!]^H\epsilon\left[\kappa_1 \mapsto \nu f^{\phi_1[\phi'/\kappa],h}_{\kappa_1,\epsilon}\right]\right]\delta\\ &= \mathbb{M}[\![\phi_1]\!]^H\epsilon[\kappa \mapsto \mathbb{M}[\![\phi']\!]^H\epsilon]\left[\kappa_1 \mapsto \nu f^{\phi_1,h}_{e[\kappa\mapsto\mathbb{M}[\![\phi']\!]^H\epsilon],\kappa_1}\right]\delta\\ &= \mathbb{M}[\![\nu\kappa_1.\phi_1]\!]^H\epsilon[\kappa \mapsto \mathbb{M}[\![\phi']\!]^H\epsilon]\delta\\ &= f^{\nu\kappa_1.\phi_1}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]\epsilon)\delta\end{aligned}$$

—$\phi = \langle\mathcal{A}\rangle\phi_1$:

$$\begin{aligned}\mathbb{M}[\![\phi[\phi'/\kappa]]\!]^H &= f_{\mathcal{A}}(\mathbb{M}[\![\phi_1[\phi'/\kappa]]\!]^H\epsilon)\delta\\ &= f_{\mathcal{A}}\left(f^{\phi_1}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]e)\right)\delta\\ &= f^{\langle\mathcal{A}\rangle\phi_1}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]e)\delta\end{aligned}$$

—$\phi = \neg\phi_1$:

$$\begin{aligned}\mathbb{M}[\![\phi[\phi'/\kappa]]\!]^H\epsilon\delta &= \mathit{Net} - \mathbb{M}[\![\phi_1[\phi'/\kappa]]\!]^H\epsilon\delta\\ &= \mathit{Net} - f^{\phi_1}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]e)\delta\\ &= f^{\neg\phi_1}_{\kappa,\epsilon}(\mathbb{M}[\![\phi']\!]e)\delta \quad \square\end{aligned}$$

THEOREM 4.8. *If there exists a proof* Π *for* $H \vdash N : \phi$ *then* $N \in \mathbb{M}[\![\phi]\!]^H\epsilon_0\delta_0$.

PROOF. The theorem can be proved by induction on the length of Π. The proof follows the schema of the proof for Theorem 3.13.

Base of Induction. If the length of Π is 0 then we can have:

—$\phi = \mathbf{tt}$.

—$\phi = \nu\kappa.\phi'$ and $\exists N'.N' : \nu\kappa.\phi' \in H$ and $N' \equiv N$;

—$\phi = \neg\langle\mathcal{A}\rangle\phi'$, and $\forall(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!] \nexists N'.N \succ\!\xrightarrow{a}\! N'$;

—$\phi =$ tp@s and there exists et such that $N \xrightarrow{et@s}\!\!\!\!\rightarrow N'$ and et : tp;

—$\phi = \neg$ tp@s and for all et such that $N \xrightarrow{et@s}\!\!\!\!\rightarrow N'$ then $\neg(et$: tp).

For each of these cases we have that $N \in \mathbb{M}[\![\phi]\!]^H$.

Inductive Hypothesis. We suppose that if $H \vdash N : \phi$ is provable in less then n steps then $N \in \mathbb{M}[\![\phi]\!]^H$.

Inductive Step. Let $H \vdash N : \phi$ provable in $n+1$ steps. We have to consider different cases depending on the last applied rule. All cases are a generalization of the Theorem 3.13, the only new cases are $R6$ and $R5$. We have to prove that

$$\mathbb{M}[\![\nu\kappa.\phi]\!]^H \epsilon\delta = \mathbb{M}[\![\phi[\nu\kappa.\phi/\kappa]]\!]^{H \cup N:\nu\kappa.\phi\{\delta\}} \epsilon\delta$$

and no formulae in H involve $\nu\kappa.\phi$ as strict subformula and $N : \nu\kappa.\phi \notin H$.

For Lemma 4.7 we have that

$$\mathbb{M}[\![\phi[\nu\kappa.\phi/\kappa]]\!]^{H \cup N:\nu\kappa.\phi\{\delta\}} e\delta = f^{\phi,h}_{\kappa,\epsilon}\left(\mathbb{M}[\![\nu\kappa.\phi]\!]^{H \cup N:\nu\kappa.\phi\{\delta\}}\epsilon\right)$$

where $h\delta' = \{N' | N' : \nu\kappa.\phi\delta' \in H \cup \{N : \nu\kappa.\phi\{\delta\}\}\}$. From the definition of $\mathbb{M}[\![\cdot]\!]^H$ we have that

$$\mathbb{M}[\![\nu\kappa.\phi]\!]^{H \cup N:\nu\kappa.\phi}\epsilon = \nu f^{\phi,h}_{\kappa,\epsilon} \cup h$$

then we have

$$\mathbb{M}[\![\phi[\nu\kappa.\phi/\kappa]]\!]^{H \cup N:\nu\kappa.\phi\{\delta\}}\epsilon\delta = f^{\phi,h}_{\kappa,\epsilon}\left(\nu f^{\phi,h}_{\kappa,\epsilon} \cup h\right) = \nu f^{\phi,h}_{\kappa,\epsilon}\delta$$

From the inductive hypothesis we have that $N \in \nu f^{\phi,h}_{\kappa,\epsilon}\delta$ then

$$h' = \lambda\delta'.\{N | \nu\kappa.\phi\{\delta\} = \nu\kappa.\phi\{\delta'\}\} \subseteq \nu f^{\phi,h}_{\kappa,\epsilon}$$

and $h = h_1 \cup h'$, where $h_1\delta' = \{N | N : \phi\{\delta'\} \in H\}$. For the previous lemma we have that

$$\nu f^{\phi,h}_{\kappa,\epsilon}\delta = \nu f^{\phi,h_1}_{\kappa,\epsilon} = \mathbb{M}[\![\nu\kappa.\phi]\!]^H\epsilon\delta$$

then the thesis follows. □

4.3.2 *Finiteness of Proofs.* We want to now prove that if N is such that the set of nets that are reachable from N, quotiented with respect to relation $\equiv$, is finite then the sequents $H \vdash N : \phi$, for every formula ϕ and hypothesis H, have only finite proofs.

First, we introduce some technical tools. Given two sequents π_1 and π_2 we define $\pi_1 \sqsubseteq_I \pi_2$ as the least relation satisfying:

—$H \vdash N : \phi_1 \vee \phi_2 \sqsubseteq_I H \vdash N : \phi_i$;
—$H \vdash N : \langle\mathcal{A}\rangle\phi \sqsubseteq_I H \vdash N' : \phi'$, where $N \overset{a}{\rightarrowtail} N'$, $(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!]$ and $\phi' = \phi\{\delta\}$;
—$H \vdash N : \phi_1 \vee \phi_2 \sqsubseteq_I H \vdash N : \phi_i$;
—if $N : \nu\kappa.\phi \notin H$ then $H \vdash N : \nu\kappa.\phi \sqsubseteq_I H' \cup N : \nu\kappa.\phi \vdash N : \phi[\nu\kappa.\phi/\kappa]$, where

$$H' = H - \left\{N' : \phi' | \nu\kappa.\phi \prec \phi'\right\}$$

—if $H_1 \vdash N_1 : \phi_1 \sqsubseteq_I H_2 \vdash N_2 : \phi_2$ then $H_1 \vdash N_1 : \neg\phi_1 \sqsubseteq_I H_2 \vdash N_2 : \neg\phi_2$.

We also define $\sqsubseteq$ as the transitive closure of $\sqsubseteq_I$.

We denote with $\{\pi_i\}_{i\in I}$, where $I \subset \mathbb{N}$, a sequence of sequents such that, for all $i, j \in I$, if $i < j$ then $\pi_i \sqsubseteq \pi_j$, we also use H_i, N_i and ϕ_i for the set of hypotheses, net and formula in π_i ($\pi_i = H_i \vdash N : \phi_i$).

Definition 4.9. Let $\mathcal{N}$ be a set of nets; we define $\mathcal{A}(\mathcal{N})$ as follows:

$$\mathcal{A}(\mathcal{N}) = \{a | \exists N_1, N_2 \in \mathcal{N} : \ N_1 \stackrel{a}{\rightarrowtail} N_2\}$$

Definition 4.10. Let $\mathcal{N}$ be a set of nets; we define $\mathcal{R}_{\mathcal{N}}(\phi)$ as follows:

—$\mathcal{R}_{\mathcal{N}}(\ \mathtt{tp@}s) = \{\ \mathtt{tp@}s\}$;
—$\mathcal{R}_{\mathcal{N}}(\neg\ \mathtt{tp@}s) = \{\neg\ \mathtt{tp@}s\}$;
—$\mathcal{R}_{\mathcal{N}}(\kappa) = \emptyset$;
—$\mathcal{R}_{\mathcal{N}}(\neg\neg\phi) = \mathcal{R}_{\mathcal{N}}(\phi) \cup \{\neg\neg\phi\}$;
—$\mathcal{R}_{\mathcal{N}}(\langle\mathcal{A}\rangle\phi) = \bigcup_{a\in\mathcal{A}(\mathcal{N})} \bigcup_{(a;\delta)\in\mathbb{A}[\![\mathcal{A}]\!]} \mathcal{R}_{\mathcal{N}}(\phi\{\delta\}) \cup \{\langle\mathcal{A}\rangle\phi\}$
—$\mathcal{R}_{\mathcal{N}}(\nu\kappa.\phi) = \mathcal{R}_{\mathcal{N}}(\phi)[\nu\kappa.\phi/\kappa] \cup \{\nu\kappa.\phi\}$;
—$\mathcal{R}_{\mathcal{N}}(\neg\nu\kappa.\phi) = \mathcal{R}_{\mathcal{N}}(\neg\phi)[\nu\kappa.\phi/\kappa] \cup \{\neg\nu\kappa.\phi\}$;
—$\mathcal{R}_{\mathcal{N}}(\phi_1 \vee \phi_2) = \mathcal{R}_{\mathcal{N}}(\phi_1) \cup \mathcal{R}_{\mathcal{N}}(\phi_2) \cup \{\phi_1 \cup \phi_2\}$;
—$\mathcal{R}_{\mathcal{N}}(\neg(\phi_1 \vee \phi_2)) = \mathcal{R}_{\mathcal{N}}(\neg\phi_1) \cup \mathcal{R}_{\mathcal{N}}(\neg\phi_2) \cup \{\neg(\phi_1 \cup \phi_2)\}$;

LEMMA 4.11. *For every finite set of nets $\mathcal{N}$ and for every formula ϕ $\mathcal{R}_{\mathcal{N}}(\phi)$ is finite.*

PROOF. We prove the lemma by induction on the structure of ϕ.

Base of Induction. If $\phi = \ \mathtt{tp@}s$, $\phi = \neg\ \mathtt{tp@}s$ or $\phi = \kappa$ then $\mathcal{R}_{\mathcal{N}}(\phi)$ is finite.

Inductive Hypothesis. We suppose that for every formulae ϕ_1 and ϕ_2 if $\mathcal{N}$ is finite then $\mathcal{R}_{\mathcal{N}}(\phi_1)$ and $\mathcal{R}_{\mathcal{N}}(\phi_2)$ are also finite.

Inductive Step.

—$\phi = \phi_1 \vee \phi_2$, by definition of $\mathcal{R}_{\mathcal{N}}$ we have that
$$\mathcal{R}_{\mathcal{N}}(\phi_1 \vee \phi_2) = \{\phi_1 \vee \phi_2)\} \cup \mathcal{R}_{\mathcal{N}}(\phi_1) \cup \mathcal{R}_{\mathcal{N}}(\phi_2)$$
from the inductive hypothesis we have that $\mathcal{R}_{\mathcal{N}}(\phi_1)$ and $\mathcal{R}_{\mathcal{N}}(\phi_2)$ are finite, then $\mathcal{R}_{\mathcal{N}}(\phi_1 \vee \phi_2)$ is finite;
—$\phi = \langle\mathcal{A}\rangle\phi_1$, we have that
$$\mathcal{R}_{\mathcal{N}}(\langle\mathcal{A}\rangle\phi) = \bigcup_{a\in\mathcal{A}(\mathcal{N})} \bigcup_{(a;\delta)\in\mathbb{A}[\![\mathcal{A}]\!]} \mathcal{R}_{\mathcal{N}}(\phi\{\delta\}) \cup \{\langle\mathcal{A}\rangle\phi\}$$
which is finite because:
—$\mathcal{R}_{\mathcal{N}}(\phi_1\{\delta\}$ is finite from the inductive hypothesis;
—and $\mathcal{A}(\mathcal{N})$ is finite if $\mathcal{N}$ is finite.
—$\phi = \nu\kappa'.\phi_1$, in this case we have that:
$$\mathcal{R}_{\mathcal{N}}(\nu\kappa'.\phi_1) = \mathcal{R}_{\mathcal{N}}(\phi_1)[\nu\kappa.\phi/\kappa'] \cup \{\nu\kappa.\phi_1\}$$
which is finite from the inductive hypothesis.
—$\phi = \neg\neg\phi_1$, in this case $\mathcal{R}_{\mathcal{N}}(\neg\neg\phi_1) = \{\neg\neg\phi_1\} \cup \mathcal{R}_{\mathcal{N}}(\phi_1)$ which is finited because of the inductive hypothesis.
—the thesis follows analogously for the cases $\neg\langle\mathcal{A}\rangle\phi_1$, $\neg(\phi_1 \vee \phi_2)$ and $\neg\nu\kappa.\phi_1$. □

LEMMA 4.12. *Let ϕ be a closed formula; then for all ϕ' and δ:*

$$\phi'[\phi/\kappa]\{\delta\} = \phi'\{\delta\}[\phi/\kappa]$$

PROOF. The lemma is proved by induction on the syntax of ϕ'.

Base of Induction. If $\phi' = \kappa, \phi' = \kappa'$ with $\kappa \neq \kappa'$ or $\phi' = \mathtt{tp}@s$ then $\phi'[\phi/\kappa]\{\delta\} = \phi'\{\delta\}[\phi/\kappa]$

Inductive Hypothesis. Let ϕ_1 and ϕ_2 be such that for every δ ($i \in \{1,2\}$):

$$\phi_i[\phi/\kappa]\{\delta\} = \phi_i\{\delta\}[\phi/\kappa]$$

Inductive Step

—$\phi' = \phi_1 \wedge \phi_2$:

$$\begin{aligned}\phi'[\phi/\kappa]\{\delta\} &= (\phi_1 \wedge \phi_2)[\phi/\kappa]\{\delta\}\\ &= \phi_1[\phi/\kappa]\{\delta\} \wedge \phi_1[\phi/\kappa]\{\delta\}\\ &= \phi_1\{\delta\}[\phi/\kappa] \wedge \phi_1\{\delta\}[\phi/\kappa]\\ &= (\phi_1 \wedge \phi_2)\{\delta\}[\phi/\kappa]\end{aligned}$$

—$\phi' = \langle\mathcal{A}\rangle\phi_1$:

$$\begin{aligned}\phi'[\phi/\kappa]\{\delta\} &= (\langle\mathcal{A}\rangle\phi_1)[\phi/\kappa]\{\delta\}\\ &= \langle\mathcal{A}\{\delta\}\rangle(\phi_1[\phi/\kappa]\{\delta'\})\\ &= \langle\mathcal{A}\{\delta\}\rangle(\phi_1\{\delta'\}[\phi/\kappa])\\ &= (\langle\mathcal{A}\{\delta\}\rangle\phi_1\{\delta'\})[\phi/\kappa]\\ &= (\langle\mathcal{A}\rangle\phi_1)\{\delta\}[\phi/\kappa]\end{aligned}$$

where $\delta'(u) = \begin{cases} u & \text{if } ?u \text{ appears in } \mathcal{A} \\ \delta(u) & \text{otherwise} \end{cases}$.

—$\phi' = \neg\phi_1$:

$$\begin{aligned}\phi'[\phi/\kappa]\{\delta\} &= (\neg\phi_1)[\phi/\kappa]\{\delta\}\\ &= \neg(\phi_1[\phi/\kappa]\{\delta\})\\ &= \neg(\phi_1\{\delta\}[\phi/\kappa])\\ &= (\neg\phi_1)\{\delta\}[\phi/\kappa]\end{aligned}$$

—$\phi' = \nu\kappa'.\phi_1$: if $\kappa = \kappa'$ then $\phi'[\phi/\kappa]\{\delta\} = \phi'\{\delta\} = \phi'\{\delta\}[\phi/\kappa]$ else, if $\kappa \neq \kappa'$:

$$\begin{aligned}\phi'[\phi/\kappa]\{\delta\} &= (\nu\kappa'\phi_1)[\phi/\kappa]\{\delta\}\\ &= \nu\kappa'(\phi_1[\phi/\kappa]\{\delta\})\\ &= \nu\kappa'(\phi_1\{\delta\}[\phi/\kappa])\\ &= (\nu\kappa'\phi_1)\{\delta\}[\phi/\kappa]\end{aligned}$$

□

LEMMA 4.13. *Let ϕ and ϕ' be formulae and let $\mathcal{N}$ be a set of nets, if ϕ' is closed then:*

$$\mathcal{R}_{\mathcal{N}}(\phi[\phi'/\kappa]) \subseteq \mathcal{R}_{\mathcal{N}}(\phi)[\phi'/\kappa] \cup \mathcal{R}_{\mathcal{N}}(\phi')$$

PROOF. We prove the lemma by induction on the structure of ϕ.

Base of Induction. If $\phi = \mathtt{tp}@s, \phi = \neg\,\mathtt{tp}@s$ or $\phi = \kappa$ the thesis follows easily.

Inductive Hypothesis. We suppose that for ϕ_1 and ϕ_2 it holds that (for $i \in \{1,2\}$).

$$\mathcal{R}_{\mathcal{N}}(\phi_i[\phi'/\kappa]) \subseteq \mathcal{R}_{\mathcal{N}}(\phi_i)[\phi'/\kappa]) \cup \mathcal{R}_{\mathcal{N}}(\phi')$$

Inductive Step. We are going to prove for $\phi = \phi_1 \vee \phi_2$, $\phi = \langle \mathcal{A} \rangle \phi$ and $\phi = \nu\kappa.\phi'$. Cases $\phi = \neg(\phi_1 \vee \phi_2)$, $\phi = \neg\langle \mathcal{A} \rangle \phi$ and $\phi = \neg\nu\kappa.\phi'$ can be obtained analogously.

—$\phi = \phi_1 \vee \phi_2$, we have that

$$\begin{aligned}\mathcal{R}_{\mathcal{N}}((\phi_1 \vee \phi_2)[\phi'/\kappa]) &= \mathcal{R}_{\mathcal{N}}(\phi_1[\phi'/\kappa] \vee \phi_2[\phi'/\kappa]) \\ &= \mathcal{R}_{\mathcal{N}}(\phi_1[\phi'/\kappa]) \cup \mathcal{R}_{\mathcal{N}}(\phi_2[\phi'/\kappa]) \cup \{\phi_1 \vee \phi_2\}\end{aligned}$$

from the inductive hypothesis we also have that

$$\mathcal{R}_{\mathcal{N}}(\phi_i[\phi'/\kappa]) \subseteq \mathcal{R}_{\mathcal{N}}(\phi_i)[\phi'/\kappa]) \cup \mathcal{R}_{\mathcal{N}}(\phi')$$

then we have that

$$\begin{aligned}&\mathcal{R}_{\mathcal{N}}(\phi_1[\phi'/\kappa]) \cup \mathcal{R}_{\mathcal{N}}(\phi_2[\phi'/\kappa]) \cup \{\phi_1 \vee \phi_2\} \subseteq \\ &\qquad \{\phi_1[\phi'/\kappa] \vee \phi_2[\phi'/\kappa]\} \cup \mathcal{R}_{\mathcal{N}}(\phi_1)[\phi'/\kappa] \cup \mathcal{R}_{\mathcal{N}}(\phi_2)[\phi'/\kappa] \cup \mathcal{R}_{\mathcal{N}}(\phi') \\ &\qquad = \mathcal{R}_{\mathcal{N}}(\phi_1 \vee \phi_2)[\phi'/\kappa] \cup \mathcal{R}_{\mathcal{N}}(\phi')\end{aligned}$$

that is the thesis.

—$\phi = \langle \mathcal{A} \rangle \phi_1$, then

$$\mathcal{R}_{\mathcal{N}}(\langle \mathcal{A} \rangle \phi_1[\phi'/\kappa]) = \bigcup_{a \in \mathcal{A}(\mathcal{N})} \bigcup_{(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!]} \mathcal{R}_{\mathcal{N}}(\phi_1[\phi'/\kappa]\{\delta\}) \cup \{\langle \mathcal{A} \rangle \phi[\phi'/\kappa]\} = (*)$$

For the previous lemma, we have that

$$\phi_1[\phi'/\kappa]\{\delta\} = \phi_1\{\delta\}[\phi'/\kappa]$$

then

$$\begin{aligned}(*) &= \textstyle\bigcup_{a \in \mathcal{A}(\mathcal{N})} \bigcup_{(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!]} \mathcal{R}_{\mathcal{N}}(\phi_1\{\delta\}[\phi'/\kappa]) \cup \{\langle \mathcal{A} \rangle \phi[\phi'/\kappa]\} \\ &= \textstyle\bigcup_{a \in \mathcal{A}(\mathcal{N})} \left(\bigcup_{(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!]} \mathcal{R}_{\mathcal{N}}(\phi_1\{\delta\})[\phi'/\kappa] \cup \mathcal{R}_{\mathcal{N}}(\phi')\right) \cup \{\langle \mathcal{A} \rangle \phi[\phi'/\kappa]\} \\ &= \textstyle\bigcup_{a \in \mathcal{A}(\mathcal{N})} \left(\bigcup_{(a;\delta) \in \mathbb{A}[\![\mathcal{A}]\!]} \mathcal{R}_{\mathcal{N}}(\phi_1\{\delta\})[\phi'/\kappa]\right) \cup \mathcal{R}_{\mathcal{N}}(\phi') \cup \{\langle \mathcal{A} \rangle \phi[\phi'/\kappa]\} \\ &= \mathcal{R}_{\mathcal{N}}(\langle \mathcal{A} \rangle \phi)[\phi'/\kappa] \cup \mathcal{R}_{\mathcal{N}}(\phi')\end{aligned}$$

—$\phi = \nu\kappa'.\phi_1$, we have that:

$$\mathcal{R}_{\mathcal{N}}(\nu\kappa'.\phi_1[\phi'/\kappa]) = \mathcal{R}_{\mathcal{N}}(\phi_1[\phi'/\kappa])[\phi_1/\kappa'] \cup \{\nu\kappa.\phi_1[\phi'/\kappa]\}$$

from the inductive hypothesis it follows that

$$\mathcal{R}_{\mathcal{N}}(\phi_1[\phi'/\kappa])[\phi_1/\kappa'] = \mathcal{R}_{\mathcal{N}}(\phi_1)[\phi'/\kappa][\phi_1/\kappa'] \cup \mathcal{R}_{\mathcal{N}}(\phi')[\phi_1/\kappa]$$

for hypothesis ϕ' is closed then

$$\mathcal{R}_{\mathcal{N}}(\phi_1[\phi'/\kappa])[\phi_1/\kappa'] = \mathcal{R}_{\mathcal{N}}(\phi_1)[\phi_1/\kappa'][\phi'/\kappa] \cup \mathcal{R}_{\mathcal{N}}(\phi') \qquad \square$$

COROLLARY 4.14. *Let $\nu\kappa.\phi$ be a formula and $\mathcal{N}$ a set of nets then:*

$$\mathcal{R}_{\mathcal{N}}(\phi[\nu\kappa.\phi/\kappa]) = \mathcal{R}_{\mathcal{N}}(\phi)[\nu\kappa.\phi/\kappa] \cup \{\nu\kappa.\phi\}$$

COROLLARY 4.15. *Let N be a net and let $\mathcal{N} = \{N' | N \succ\!\!\rightarrow^* N'\}$. Let $\pi_1 = H_1 \vdash N_1 : \phi_1$ and $\pi_2 = H_2 \vdash N_2 : \phi_2$ such that: $N_1, N_2 \in \mathcal{N}$ and $\pi_1 \sqsubseteq_I \pi_2$, then:*

$$\mathcal{R}_{\mathcal{N}}(\phi_2) \subseteq \mathcal{R}_{\mathcal{N}}(\phi_1)$$

Moreover, for every formula ϕ and every set of hypothesis H:

$$\{\phi' | \exists H', N'.H \vdash N : \phi \sqsubseteq H' \vdash N' : \phi'\} \subseteq \mathcal{R}_{\mathcal{N}}(\phi)$$

LEMMA 4.16. *Let N be a net such that the set $\mathcal{N} = \{N' | N \succ\!\!\rightarrow^* N'\}/\equiv$ is finite, ϕ be a closed formula, and H be a set of hypotheses, then every sequence $\{\pi_i\}_{i \in I}$ starting from $\pi = H \vdash N : \phi$ is finite.*

PROOF. We have that the set of formulae that appear in the sequence is finite. Indeed, it is a subset of $\mathcal{R}_{\mathcal{N}}(\phi)$, which is finite for Lemma 4.11. Let

$$\Phi = \{\phi' | \exists i.\pi_i = H' \vdash N' : \phi'\}$$

We prove the theorem by induction on the size n of Φ.

Base of Induction. Let $n = 1$, and let ϕ then be the only formula that appears in Π. We can distinguish two cases:

—$\phi = \texttt{tp@}s, \phi = \neg\,\texttt{tp@}s, \phi = tt$ or $\phi = \nu\kappa.\phi$ or $\phi = \neg\nu\kappa.\phi$ and $N : \nu\kappa.\phi \in H$. In this case there are no sequents π' such that $\pi \sqsubseteq_I \pi'$. Every sequence starting from π is obviously finite.

—$\phi = \nu\kappa.\kappa$, then for every N and H the only (finite) sequence from $H \vdash N : \phi$ is the following:

$$H \vdash N : \nu\kappa.\kappa \sqsubseteq_I H \cup \{N : \nu\kappa.\kappa\} \vdash N : \nu\kappa.\kappa$$

Inductive Hypothesis. We suppose that if size of Φ is less or equal to n then π has only finite sequences.

Inductive Step. Let $\pi = H \vdash N : \phi$ such that size of Φ is $n + 1$. We can distinguish two cases.

—There isn't in Φ a formula like $\nu\kappa.\phi$. In this case we have that, for every i, ϕ_{i+i} is a strict subterm of ϕ_i (up to a substitution). This means that every formula occurs exactly one time in the sequence. For the inductive hypothesis we have that the sequence that starts from π_1 is finite. Then also the sequence that starts from π_0 is finite.

—There exists $\nu\kappa_1.\phi_1$ in Φ. Let $\nu\kappa_1.\phi_1$ be such that: $\forall\phi \in \Phi : \phi' \not\prec \nu\kappa.\phi_1$. Let π_i be the first sequence in which occur $\nu\kappa.\phi_1$. For every $\pi_j = H_j \vdash N_j : \phi_j$, such that $\pi_i \sqsubseteq \pi_j$, $N_i : \nu\kappa.\phi_1 \in H_j$ Hence, the size of $\{\pi_j = H_j \vdash N_j : \phi_j | j \in I \,\&\, \pi_i \sqsubseteq \pi_j\}$ is less than or equal to the size of $\mathcal{N}$. Let $\pi_{\hat{j}} = H_{\hat{j}} \vdash N_{\hat{j}} : \nu\kappa.\phi_1$ be its greatest element. If $\pi_{\hat{j}}$ is a successful sequence, then the sequence is finite; otherwise there exists π' such that $\pi_{\hat{j}} \sqsubseteq_I \pi'$ and

$$|\{\phi' | \exists i.\pi' \sqsubseteq_I \pi_i = H' \vdash N' : \phi'\}| \leq n$$

then, from the inductive hypothesis, every sequence from π' is finite then also $\{\pi_i\}_{i\in I}$ is finite. □

THEOREM 4.17. *For every net N such that the set $\mathcal{N} = \{N' | N \rightarrowtail^* N'\}/\equiv$ is finite, for every closed formula ϕ and for every set of hypothesis H, the proof for $H \vdash N : \phi$ is finite.*

PROOF. We have that $\{\pi | H \vdash N : \phi \sqsubseteq_I \pi\}$ is finite (indeed $\mathcal{N}$ is finite), then if Π is an infinite proof for $H \vdash N : \phi$, there exists an infinite sequence $\{\pi_i\}_{i\in I}$ from $H \vdash N : \phi$ that is in contradiction with Lemma 4.16 □

Definition 4.18. Let N be such that the set $\mathcal{N} = \{N' | N \rightarrowtail^* N'\}/equiv$ is finite, and let be a formula and it be a set of hypotheses. We define $\min(H \vdash N : \phi)$ as the least length of a proof Π for the sequent $H \vdash N : \phi$.

4.3.3 *Completeness.* In order to prove the completeness, first we prove that, if the set of reachable nets from N is finite either $H \vdash N : \phi$ or $H \vdash N : \neg\phi$ is provable.

LEMMA 4.19. *Let N be such that it has a finite set of reachable nets, then either the sequent $H \vdash N : \phi$ or the sequent $H \vdash N : \neg\phi$ is provable.*

PROOF. The lemma is provable by induction on $\min(H \vdash N : \phi)$ as the Lemma 3.14. □

Please, notice that if the set of nets reachable from N is not finite then the previous lemma does not hold. Indeed, we are not able to prove properties like $\nu\kappa.\phi \wedge [\circ]\kappa$ for which we need to *explore* all the reachable nets.

THEOREM 4.20. *Let N be such that the set of reachable nets is finite and let ϕ be a formula. If $N \in \mathbb{M}[\![\phi]\!]^H \epsilon_0 \delta_0$ then $H \vdash N : \phi\{\delta\}$ is provable.*

PROOF. We suppose that $N \in \mathbb{M}[\![\phi]\!]^H \epsilon\delta$ and that $H \vdash N : \phi\{\delta\}$ is not provable. For the previous lemma we have that $H \vdash N : \neg\phi\{\delta\}$ is provable, and from the soundness we have also that $N \in \mathbb{M}[\![\neg\phi]\!]^H \epsilon\delta$. Thus from the definition of $\mathbb{M}[\![\cdot]\!]^H$ we have that $N \in Net - \mathbb{M}[\![\phi]\!]^H \epsilon\delta$, which is in contradiction with the hypothesis. □

4.3.4 *Completeness for Infinite Systems.* In the previous section we showed that completeness for the proof system can be guaranteed only for a subset of the KLAIM nets.

In this section, we present a subset of formulae for which we are able to prove a completeness result. These formulae, which we call *regular*, are those that permit specifying *eventually-properties*.

In order to simplify definition of such formulae we will explicitly use in the syntax formulae like $\phi_1 \wedge \phi_2$, $[\mathcal{A}]\phi$ and $\mu\kappa.\phi$. In previous sections we have shown how to express these formulae within our original language; indeed they can be thought of as macro. Consequently we have to modify the rules $R1$, $R5$ and $R6$ in Table XV as follows:

$$R1 \qquad \frac{H \vdash N : \phi_1 \qquad H \vdash N : \phi_2}{H \vdash N : \phi_1 \wedge \phi_2}$$

$$R5 \qquad \frac{H \vdash N_1 : \phi\{\delta_1\} \quad H \vdash N_2 : \phi\{\delta_2\} \quad \ldots}{H \vdash N : [\mathcal{A}]\phi} \left[\begin{array}{l} \forall (a_i; \delta_i) \in \mathbb{A}[\![\mathcal{A}]\!] : \\ \quad \forall N_i \in \left\{ N' | N \xrightarrow{a_i} N' \right\} \end{array} \right]$$

$$R6 \qquad \frac{H' \cup \{N : \mu\kappa.\phi\} \vdash N : \phi[\mu\kappa.\phi/\kappa]}{H \vdash N : \mu\kappa.\phi} [N' \equiv N, N' : \mu\kappa.\phi \notin H]$$

Definition 4.21. We say that a formula ϕ is *regular* if it is generated by:

$$\phi \ ::= \ \mathbf{tt} \mid \mathbf{ff} \mid \mathtt{tp}@\sigma \mid \neg\,\mathtt{tp}@\sigma \mid \langle\mathcal{A}\rangle\phi \mid [\mathcal{A}]\phi \mid \phi \vee \phi \mid \phi_1 \wedge \phi_2 \mid \kappa \mid \mu\kappa.\phi$$

LEMMA 4.22. *If ϕ is regular and it does not have a subformula $\mu\kappa.\phi'$ then for every net N if $N \in [\![\phi]\!]^H$, $H \vdash N : \phi$ is provable.*

PROOF. The lemma follows directly from the completeness of formulae without recursion. □

Definition 4.23. Let $\mu\kappa.\phi$ be a closed formula. We define $\widetilde{\phi}_i$, as follows:

$$\widetilde{\phi}_0 = \mathbf{ff}$$

$$\widetilde{\phi}_{i+1} = \phi[\widetilde{\phi}_i/\kappa]$$

LEMMA 4.24. *Let $\mu\kappa.\phi$ be a closed formula and let N be a net:*

$$N \in \mathbb{M}[\![\mu\kappa.\phi]\!] \Leftrightarrow \exists i.\ N \models \widetilde{\phi}_i$$

LEMMA 4.25. *For every formula ϕ' ($\phi' \neq \kappa$), and $\mu\kappa.\phi$: for every i if $H \vdash N : \phi'[\widetilde{\phi}_i/\kappa]$ is provable then $H \vdash N : \phi'[\mu\kappa.\phi/\kappa]$ is provable too.*

PROOF. We prove the lemma by induction on the length n of the proof for $H \vdash N : \phi'[\widetilde{\phi}_i/\kappa]$.

Base of Induction. $n = 0$ in this case $N : \phi'[\widetilde{\phi}_i/\kappa]$ is a successful sequent. Then $\phi'[\widetilde{\phi}/\kappa] = \mathtt{tp}@s$, $\phi'[\widetilde{\phi}/\kappa] = \neg\,\mathtt{tp}@s$. In all cases also $H \vdash N : \phi'[\mu\kappa.\phi/\kappa]$ is a successful sequent.

Base of Induction. We suppose that, for every formula ϕ', if $H \vdash N : \phi'[\widetilde{\phi}_i/\kappa]$ is provable in at most n steps then $H \vdash N : \phi'[\mu\kappa.\phi/\kappa]$ is also provable.

Inductive Step. Let $H \vdash N : \phi'[\widetilde{\phi}_i/\kappa]$ be provable in $n+1$ steps. We distinguish according to the first applied rule.

—$R1$ In this case we have that:

$$\frac{H \vdash N : \phi_1[\widetilde{\phi}/\kappa] \qquad H \vdash N : \phi_2[\widetilde{\phi}/\kappa]}{H \vdash N : \phi_1[\widetilde{\phi}_i/\kappa] \wedge \phi_2[\widetilde{\phi}_i/\kappa])}$$

We have that if $\phi_i \neq \kappa$ then the thesis follows for the inductive hypothesis, otherwise $\phi_1[\widetilde{\phi}_i/\kappa] = \phi[\phi_{i-1}/\kappa]$ (i has to be greatest then 0), which is provable with at most n steps. By applying the inductive hypothesis we have that $(\phi_1 \wedge \phi_2)[\mu\kappa.\phi/\kappa]$ is also provable.

—for $R2$, $R3$ and $R5$[1] we can proceed as for the previous case.

—$R6$ In this case we have that:

$$\frac{H' \cup \{N : \mu\kappa'\phi'[\widetilde{\phi}_i/\kappa]\} \vdash N : \phi'[\mu\kappa'.\phi'/\kappa'][\widetilde{\phi}/\kappa]}{H \vdash N : \mu\kappa'.\phi'[\widetilde{\phi}_i/\kappa]}$$

by using the inductive hypothesis we have that

$$H' \cup \{N : \mu\kappa'\phi'[\widetilde{\phi}_i/\kappa]\} \vdash N : \phi'[\mu\kappa'.\phi'/\kappa'][\mu\kappa.\phi/\kappa]$$

is provable. □

THEOREM 4.26. *If ϕ is regular then, for every net N if $N \in [\![\phi]\!]^H$, $H \vdash N : \phi$ is provable.*

PROOF. The proof follows by using the previous two lemmata. □

[1]If ϕ is regular rule $R4$ will never be applied in any proof for ϕ.

5. USING THE LOGIC

In this section, we will present three examples that should help convince the reader of the qualities of the proposed logic. The first example, which was presented in Section 2.2, shows how to prove properties for a simple KLAIM net. The second shows how the logic can be used for specifying the access policies of KLAIM nets. The final one formalizes properties for a more complex system managing distributed information.

5.1 The Itinerant Agent

In this section we present a simple proof built by using our proof system. Let N be the net defined as follows:

$$s_1 :: Proc_1 | \mathbf{out}(s_2) \parallel s_2 :: \mathbf{nil}$$

where

$$Proc_1 = \mathbf{in}(!u)@\mathtt{self}.\mathbf{eval}(Proc_1)@u.\mathbf{out}(\mathtt{self})@u.\mathbf{nil}$$

In Section 2.2 we have presented the computation of this net.

For this net we can prove, by using the proof systems, that:

never will tuple (s_1) *be at site* s_2 *and, at the same time,* (s_2) *be at site* s_1

This property can be formally stated as follows:

$$\phi = \nu\kappa.\ \big((\neg(s_1)@s_2) \vee (\neg(s_2)@s_1)\big) \wedge [\circ]\kappa$$

The proof for $\vdash N : \phi$ starts with the application of rule $R7$, then rule $R1$ is applied:

$$\dfrac{\dfrac{N : \phi \vdash N : \neg(s_1)@s_2) \vee (\neg(s_2)@s_1) \qquad N : \phi \vdash N : [\circ]\phi}{N : \phi \vdash N : \big((\neg(s_1)@s_2) \vee (\neg(s_2)@s_1)\big) \wedge [\circ]\phi}\, R1}{\vdash N : \phi}\, R7$$

We can prove $N : \phi \vdash N : \neg((s_1)@s_2) \vee \neg((s_2)@s_1)$ by applying rule $R2$ and obtaining the successful sequent $N : \phi \vdash N : \neg((s_1)@s_2)$. Please notice that for each net N_i reachable from N (see Section 2.2) and for all H the sequent $H \vdash N_i : \neg((s_1)@s_2) \vee \neg((s_2)@s_1)$ is provable.

The other branch of the proof, $N : \phi \vdash N : [\circ]\phi$, proceeds with the application of rule $R5$. After this rules $R7$ and $R1$ are applied again to obtain:

$$\dfrac{\dfrac{\dfrac{N_1 : \phi, N : \phi \vdash N_1 : \neg(s_1)@s_2) \vee (\neg(s_2)@s_1) \qquad N_1 : \phi, N : \phi \vdash N_1 : [\circ]\phi}{N_1 : \phi, N : \phi \vdash N_1 : \big((\neg(s_1)@s_2) \vee (\neg(s_2)@s_1)\big) \wedge [\circ]\phi}\, R1}{N : \phi \vdash N_1 : \phi}\, R7}{N : \phi \vdash N : [\circ]\phi}\, R5$$

As in the previous case, we proceed by applying rule $R2$ for one branch and rule $R7$ for the other. By proceeding analogously on the obtained sequent we obtain the successful sequent:

$$N : \phi, N_1 : \phi, N_2 : \phi, N_3 : \phi, N_4 : \phi, N_5 : \phi, N_6 : \phi \vdash N : \phi$$

5.2 Access Right Specifications

In this section, we show how to use our logic for specifying and verifying access rights in a KLAIM net. To specify that *"no process is ever evaluated at a site s"* we can use the following formulae:

$$\nu\kappa.[\mathtt{E}(?u, \mathtt{1_P}, s)]\mathbf{ff} \wedge [\circ]\kappa$$

We can also specify dynamic access policies. Let N be a net with two sites s_1 and s_2. In that net we want to guarantee that *"s_1 doesn't insert tuples at site s_2 after s_3 has evaluated a process at s_1"*. We can formalize these properties as follows:

$$\nu\kappa.[\mathtt{E}(s_3, \mathtt{1_P}, s_1)]\left(\nu\kappa'.[\mathtt{O}(s_1, \mathtt{1_t}, s_2)]\mathbf{ff} \wedge [\circ]\kappa'\right) \wedge [\circ]\kappa$$

Composing formulae we obtain more complex access specifications. Let N' be another net with two sites s_1 and s_2. We establish that *"s_1 cannot read tuples containing a pair* (locality, process) *at s_2, and s_2 cannot evaluate any process at s_1 that reads tuples that contain a locality and puts these values at s_1"*. To specify these access rights we can use the following formula:

$$\nu\kappa.[\mathtt{R}(s_1, ?u, \mathtt{1_P}), s_2)]\mathbf{ff} \wedge [\mathtt{R}(s_1, ?l\,, \mathtt{1_P}), s_2)]\mathbf{ff} \wedge$$
$$[\mathtt{E}(s_2, \mathtt{r}(!u)@s_1 \rightarrow \mathtt{o}(u), s_1) \cup \mathtt{E}(s_2, \mathtt{i}(!u)@s_1 \rightarrow \mathtt{o}(u), s_1)]\mathbf{ff} \wedge [\circ]\kappa$$

We can also specify access rights that depend on properties of nodes. Suppose that in a net we require that each node s_1 can evaluate a process at a node s_2 only if tuple (s_1) is in the tuple space at s_2. This property can be rendered as follows:

$$\nu\kappa.[\mathtt{E}(?u_1, \mathtt{1_P}, ?u_2)](u_1)@u_2 \wedge [\circ]\kappa$$

5.3 Distributed Information Systems Management

In this section, we consider a larger example of a Distributed Information Systems management. We assume that a database is distributed over three different sites, named, Inf_i $i \in \{1, 2, 3\}$. A node, named *Manager*, manages the database system sending processes to sites for updating the information. Only one updating-process at a time can be executed at a site. For this reason, inside the tuple space of Inf_i there is the tuple "F". An updating process can be evaluated at Inf_i only if tuple "F" is in its tuple space.

The net of the distributed database is defined as follows:

$$Inf_1 ::_\rho \mathbf{out}(\text{“}F\text{”}) \parallel Inf_2 ::_\rho \mathbf{out}(\text{“}F\text{”}) \parallel Inf_3 ::_\rho \mathbf{out}(\text{“}F\text{”})$$

In the tuple space of node *Manager* there is a tuple ("G") for each node Inf_i. An updating process can be started only when at least a tuple ("G") is in the tuple space of *Manager*.

Process *StartAgent* looks for a tuple ("G"). When this is found, *StartAgent* invokes *CallUpdate*, which starts the updating procedure. By guarding *CallUpdate* in *StartAgent* with an $\mathbf{in}(\text{“}G\text{”})@\mathtt{self}$ action we guarantee that the system is deadlock free.

$$StartAgent \;=\; \mathbf{in}(\text{“}G\text{”})@\mathtt{self}.\;(CallUpdate(Inf_1, Inf_2, Inf_3) \mid StartAgent)$$

$$CallUpdate\langle u_1, u_2, u_3\rangle \;=\; \mathbf{in}(\text{“}F\text{”})@u_1.\mathbf{out}(\text{“}updating\text{”})@u_1.\\ \mathbf{eval}(Update(u_2, Update(u_3, FUpdate(Manager))))@u_1.\mathbf{nil}$$

$$Update\langle u, X\rangle \;=\; \mathbf{in}(\text{“}F\text{”})@u.\mathbf{out}(\text{“}updating\text{”})@u.\mathbf{eval}(X)@u.\\ \mathbf{in}(\text{“}updating\text{”})@\mathtt{self}.\mathbf{out}(\text{“}F\text{”})@\mathtt{self}.\mathbf{nil}$$

$$FUpdate\langle u\rangle = \mathbf{in}(\text{“}updating\text{”})@\mathtt{self}.\mathbf{out}(\text{“}F\text{”})@\mathtt{self}.\mathbf{eval}(Success)@u.\mathbf{nil}$$

$$Success \;=\; \mathbf{out}(\text{“}G\text{”})@\mathtt{self}.\mathbf{nil}$$

The manager node is defined as follows:

$$Manager \;::\; StartAgent|\mathbf{out}(Inf_1)|\mathbf{out}(Inf_2)|\mathbf{out}(Inf_3)\\ |\mathbf{out}(\text{“}G\text{”})|\mathbf{out}(\text{“}G\text{”})|\mathbf{out}(\text{“}G\text{”})$$

We would like to prove that, within the above system, if a process is evaluated at site Inf_i then no process is evaluated at the same site until the updating terminates. This property is specified by the formula ϕ_1 below:

$$\phi_1 = \nu\kappa.[\mathbf{e}(?u_1, \mathtt{1_P}, ?u_2) - \mathbf{e}(?u_3, \mathtt{1_P}, Manager)]\phi_2 \wedge [\circ]\kappa$$

where

$$\phi_2 = \nu\kappa'.[\mathbf{e}(?u, \mathtt{1_P}, u_2)]\mathbf{ff} \wedge [\mathbf{e}(u_2, \mathtt{1_P}, ?u_4)]\mathbf{tt} \vee [\circ]\kappa'$$

The property can be proved by using our proof system. However, the proof is, in some sense, *automatic* and requires a large space to be presented. For this reason we omit it.

6. CONCLUSIONS AND RELATED WORKS

In this paper we have presented a variant of HML enriched with more refined action predicates and state formulae. The logic is tailored for reasoning about properties of KLAIM systems. The proposed logic has been equipped with a proof system based on tableau. This system has been inspired by Cleaveland [1990], Stirling and Walker [1991], and Winskel [1989].

The main differences of our solution with respect to the existing ones, also based on Hennessy-Milner logic, reside in the different use of transition labels. In the *standard* approaches, even those considering value passing [Rathke and Hennessy 1997], labels are considered as *basic entities* and are characterized, inside modal operators, syntactically. In our approach transition labels are characterized by means of their *properties*.

The definition of the proof system is, in some sense, standard. However, since we have explicit notions of values, proofs of *completeness* and *soundness* the results are more complex.

There are a few papers that have tackled the problem of defining a logic for process calculi with primitives for mobility. More specifically they have considered definitions of logics for $\pi-$calculus [Milner et al. 1992] and Mobile Ambients [Cardelli and Gordon 1998].

In Milner et al. [1993] a modal logic for π-calculus has been introduced. This logic aims at capturing different equivalences between processes, and at

establishing the differences between late and early bisimulation. This logic is also based on Hennessy-Milner Logic, but it has no operators for recursion. In Dam [1996] an HML-like logic with recursion for π-calculus is presented. This logic is equipped with a tableau-based proof system. However, neither Milner et al. [1993] nor Dam [1996] consider mobility/spatial features.

In Cardelli and Gordon [2000] a modal logic for Mobile Ambients [Cardelli and Gordon 1998] has been presented; in Caires and Cardelli [2001] a variant of the Ambient logic, tailored for asynchronous π-calculus, is introduced. This very interesting logic is equipped with operators for spatial and temporal properties, and for *compositional* specification of systems properties. In Cardelli and Gordon [2001], new operators for expressing logical properties of name restrictions have been added. This logic permits describing properties of spatial configurations and of mobile computation, including security properties.

All of these properties are essentially *state based*, in the sense that systems are proved to be safe/unsafe by only taking into account spatial configurations. The structure of the system can be formalized in details: for instance one can *count* the resources in the system. However, any information about the history of an ambient in the system is lost. For instance properties like: *"an ambient can enter into n only after another one exits from n"* cannot be expressed. In Sections 5 we showed how similar properties for KLAIM systems can be expressed in our logic.

Sangiorgi [2001] shows how to express, in the ambient, logic properties like *"an ambient n is opened inside the ambient m"*. However, it is not possible to univocally identify an ambient in the system. A precise advantage of our logic is that it allows us to talk about specific, unique sites, whereas, since the ambient calculus allows multiple ambients to have the same name, it is difficult to talk about a specific ambient in the ambient logic. Moreover, to express *evolutionary properties* Sangiorgi needs to use logical operators that, in general, are not decidable. Indeed, the Ambient logic is not completely decidable and a sound and complete proof system is available only for a subset of the logic.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees and to Lorenzo Bettini and Betti Venneri, whose useful comments helped us to improve the presentation. Thanks are also due to Andy Gordon and Eugenio Moggi who, as external examiners of the Ph.D. thesis of the second author, contributed significantly to improving the paper.

REFERENCES

CAIRES, L. AND CARDELLI, L. 2001. A spatial logic for concurrency (Part I). In *Proceedings of Theoretical Aspects of Computer Software; 4th International Symposium, TACS 2001*, N. Kobayashi and B. C. Pierce, Eds. Number 2215 in Lecture Notes in Computer Science. 1–37.

CARDELLI, L. AND GORDON, A. D. 1998. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)*, M. Nivat, Ed. Lecture Notes in Computer Science, vol. 1378. sv, 140–155.

CARDELLI, L. AND GORDON, A. D. 2000. Anytime, anywhere: Modal logics for mobile ambients. In *27th Annual Symposium on Principles of Programming Languages (POPL) (Boston, MA)*. ACM.

CARDELLI, L. AND GORDON, A. D. 2001. Logical properties of name restriction. In *Proceedings of International Conference on Typed Lambda Calculi and Applications, TLCA'01*. Lecture Notes in Computer Science, vol. 2044. Springer.

CARRIERO, N. AND GELERNTER, D. 1989. Linda in Context. *Comm. ACM 32*, 10 (October), 444–458. Technical Correspondence.

CLEAVELAND, R. 1990. Tableau-based model checking in the propositional μ-calculus. *Acta Informatica 27,* 8 (September), 725–747.

DAM, M. 1996. Model checking mobile processes. *J. Info. Computation 129*, 1, 35–51.

DE NICOLA, R., FERRARI, G. L., AND PUGLIESE, R. 1998. KLAIM: A kernel language for agents interaction and mobility. *IEEE Trans. Soft. Eng. 24*, 5 (May), 315–330.

DE NICOLA, R., FERRARI, G., PUGLIESE, R., AND VENNERI, B. 2000. Types for Access Control. *Theor. Comput. Sci. 240*, 1, 215–254.

GELERNTER, D. 1985. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1, 80–112.

GELERNTER, D. 1989. Multiple Tuple Spaces in Linda. In *Proceedings, PARLE '89*, J. G. Goos, Ed. Lecture Notes in Computer Science, vol. 365. 20–27.

HENNESSY, M. AND MILNER, R. 1985. Algebraic laws for nondeterminism and concurrency. *J. ACM 32*, 1 (Jan.), 137–161.

MILNER, R. 1989. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall. SU Fisher Research 511/24.

MILNER, R., PARROW, J., AND WALKER, J. 1992. A Calculus of Mobile Processes, I and II. *Information and Computation 100,* 1, 1–40, 41–77.

MILNER, R., PARROW, J., AND WALKER, D. 1993. Modal logics for mobile processes. *Theoretical Computer Science 114*, 149–171.

RATHKE, J. AND HENNESSY, M. 1997. Local model checking for value-passing processes. In *Proceedings of International Symposium on Theoretical Aspects of Computer Software, TACS'97*, M. Abadi and T. Ito, Eds. Lecture Notes in Computer Science. Springer-Verlag, 250–265.

SANGIORGI, D. 2001. Extensionality and intensionality of the ambient logics. In *28th Annual Symposium on Principles of Programming Languages (POPL) (London, UK)*. ACM, 4–13.

STIRLING, C. AND WALKER, D. 1991. Local model checking in the modal mu-calculus. *Theor. Comput. Sci. 89*, 1 (Oct.), 161–177.

WINSKEL, G. 1989. A note on model checking the modal ν-calculus. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, Eds. Lecture Notes in Computer Science, vol. 372. Springer, Berlin, 761–772.

Received October 2001; revised May 2002; accepted June 2002

A Distributed Kripke Semantics

ROHIT CHADHA, DAMIANO MACEDONIO and VLADIMIRO SASSONE
COMPUTER SCIENCE TECHNICAL REPORT 04/2004
DEPARTMENT OF INFORMATICS
UNIVERSITY OF SUSSEX, BRIGHTON BN1 9QH, UK.
DECEMBER 2004

ABSTRACT. An intuitionistic, hybrid modal logic suitable for reasoning about distribution of resources was introduced in [10]. We extend the Kripke semantics of intuitionistic logic, enriching each possible Kripke state with a set of places, and show that this semantics is both sound and complete for the logic. In the semantics, resources of a distributed system are interpreted as atoms, and placement of atoms in a possible state corresponds to the distribution of the resources. The modalities of the logic allow us to validate properties in a *particular place*, in *some* place and in *all* places. We extend the logic with disjunctive connectives, and refine our semantics to obtain soundness and completeness for extended logic. The extended logic can be seen as an instance of *Hybrid IS5* [2, 18].

1 Introduction

In current computing paradigm, distributed resources spread over and shared amongst different nodes of a computer system is very common. For example, printers may be shared in local area networks, or distributed data may store documents in parts at different locations. The traditional reasoning methodologies are not easily scalable to these systems as they may lack implicitly trustable objects such as a central control.

This has resulted in the innovation of several reasoning techniques. A popular approach in the literature has been the use of algebraic systems such as process algebra [13, 8, 5]. These algebras have rich theories in terms of semantics [13], logics [7, 15, 4, 3], or types [8]. Another approach is logically-oriented [9, 10, 19, 14]: intuitionistic modal logics are used as foundations of type systems by exploiting the *propositions-as-types, proofs-as-programs* paradigm [6]. An instance of this was introduced in [9, 10] and the logic introduced there is the focus of our study.

The formulae in this logic include names, called *places*. Assertions in the logic are associated with places, and are validated in places. In addition to considering *whether* a formula is true, we are also interested in *where* a formula is true. The three modalities of the logic allow us to infer whether a property is

Research partially supported by '**MIKADO**: Mobile Calculi based on Domains', EU FET-GC IST-2001-32222, and '**MyThS**: Models and Types for Security in Mobile Distributed Systems', EU FET-GC IST-2001-32617.

validated in a specific place of the system ($@p$), or in an unspecified place of the system ($\Diamond$), or in any part of the system($\Box$). The modality $@p$ internalises the model in the logic and hence can be classified as a hybrid logic [1, 16, 2]. An intuitionistic natural deduction for the logic is given in [9, 10], and judgements in the logic mention the places under consideration. The natural deduction rules for $\Diamond$ and $\Box$ resemble those for existential and universal quantification of first-order intuitionistic logic.

As noted in [9, 10], the logic can also be used to reason about distribution of resources in addition to serving as the foundation of a type system. The papers [9, 10], however, lack a model to match the usage of the logic as a tool to reason about distributed resources. In this report, we bridge the gap by presenting a Kripke-style semantics [12] for the logic of [9, 10]. In Kripke-style semantics, formulae are considered valid if they remain valid when the atoms mentioned in the formulae change their value from false to true. This is achieved by using a partially ordered set of *possible states*. Informally, more atoms are true in larger states.

We extend the Kripke semantics of the intuitionistic logic [12], enriching possible states with a *fixed* set of places. In each possible state, different places satisfy different formulae. For the intuitionistic connectives, the satisfaction of formulae at a place in a possible state follows the standard definition [12]. The enrichment of the model with places reveals the true meaning of the modalities in the logic. The modality $@p$ expresses a property in a named place. The modality $\Box$ corresponds to a weak form of universal quantification and expresses a common property, and the modality $\Diamond$ corresponds to a weak form of existential quantification and expresses a property valid somewhere in the system.

In the model, we interpret atomic formulae as resources of a distributed system, and placement of atoms in a possible state corresponds to the distribution of resources. As in intuitionistic logic [12], we need not evaluate all the formulae of the language, since the interpretation follows inductively from the structure of formulae.

In order to give semantics to a logical judgment, we allow models with more places than those mentioned in the judgement. This admits the possibility that a user may be aware of only a certain subset of names in a distributed system. As we shall see, this is crucial in the proof of soundness and completeness.

In the model, we can duplicate places in a conservative way. This fact is the key to the proof of soundness of introduction of $\Box$, and the elimination of $\Diamond$. The proof of completeness follows closely the standard proof of completeness of intuitionistic logic with one important difference: in addition to witnesses for the existential ($\Diamond$), we need witnesses for the universal ($\Box$) too.

The logic in [9, 10] did not have disjunctive connectives. We extend the logic with disjunctive connectives, and refine our Kripke semantics in order to obtain

completeness. In the refined semantics, the set of places in Kripke states are not fixed. Different possible Kripke states may have *different* set of places. However, the set of places vary in a conservative way: larger Kripke states contain larger set of places.

We show that the refined semantics is both sound and complete for the extended logic. The proof of soundness once again depends on duplication of places. The proof of completeness follows closely the standard proofs of completeness of intuitionistic modal logics. The extended logic can be seen as hybridization of the well-known intuitionistic modal system *IS5* [2, 18].

The rest of the paper is organised as follows. In Section 2, we present the logic in [9, 10]. In Section 3 we present the distributed Kripke model used to interpret the logic, and prove soundness and completeness of the semantics. We present the extension of logic with logical connectives in Section4. The refined semantics is given in Section 5, where we also show soundness and completeness of the refined logic. We discuss related work in Section 6, and we summarise our results in Section 7.

2 Logic

We now introduce, through examples, the logic presented in [9, 10]. The logic is used to reason about heterogeneous distributed systems. To gain some intuition, consider a *distributed peer to peer database* where the information is partitioned over multiple communicating nodes (peers).

Informally, the database has a set of nodes, or *places*, and a set of resources (data) distributed amongst these places. The nodes are chosen from the elements of a fixed set, denoted by $p, q, r, s, \ldots$. Resources are represented by atomic formulae $A, B, \ldots \in Atoms$. Intuitively, an atom A is verified in a place p if that place can access the resource identified by A.

Were we reasoning about a particular place, the logic connectives of the intuitionistic framework would be sufficient. For example, assume that a particular document, doc, is partitioned in two parts, doc_1 and doc_2, and in order to access to the document a place has to access both of its parts. This can be formally expressed as the logical formula: $(\mathsf{doc}_1 \wedge \mathsf{doc}_2) \rightarrow \mathsf{doc}$, where $\wedge$ and $\rightarrow$ are the logical conjunction and implication. If doc_1 and doc_2 are stored in a particular place, then the usual intuitionistic rules allow to infer that the place can access the entire document.

The intuitionistic framework is extended in [10] in order to reason about different places. An assertion in such a logic takes the form "φ ***at*** p", meaning that formula φ is valid at place p. The construct "***at***" is a meta-linguistic symbol and points to the place where the reasoning is located. For example, doc_1 ***at*** p and doc_2 ***at*** p formalises the notion that the parts doc_1 and doc_2 are located at the node p. If in addition, the assertion $((\mathsf{doc}_1 \wedge \mathsf{doc}_2) \rightarrow \mathsf{doc})$ ***at*** p is valid, we

can conclude that the document doc is available at p. A formula φ may itself use three modalities to accommodate reasoning about the properties valid at different locations.

In order to internalise resources at a single location, the modality $@p$, one for every place in the system, is used. The modality $@$ casts the meta-linguistic "***at***" on the language level, and in fact the two constructs will have the same interpretation in the semantics. The modal formula $\varphi@p$ means that the property φ is valid at p, and not necessarily anywhere else. An assertion of the form $\varphi@p$ ***at*** p' means that in the place p' we are reasoning about the property φ valid at the place p. For example, suppose that the place p has got the first half of the document, i.e., doc_1 ***at*** p, and p' has got the second one, i.e., doc_2 ***at*** p'. In the logic we can formalise the fact that p' can send the part doc_2 to p by using the assertion $(\mathsf{doc}_2 \rightarrow (\mathsf{doc}_2@p))$ ***at*** p'. The rules of the logic will conclude doc_2 ***at*** p and so doc ***at*** p.

Knowing exactly where a property holds is a strong ability, and we may only know that the property holds somewhere without knowing the specific location where it holds. In order to deal with this, the logic has the $\Diamond$ modality: $\Diamond\varphi$ means that the formula φ holds in some place. In the example above, the location of doc_2 is not important as long as we know that this document is located in some place that can send it to p. Formally, this can be expressed by the formula $\Diamond(\mathsf{doc}_2 \wedge (\mathsf{doc}_2 \rightarrow (\mathsf{doc}_2@p)))$ ***at*** p'. By assuming this formula, we can infer doc_2 ***at*** p, and hence the document doc is available at p.

Even if we deal with resources distributed in heterogeneous places, we cannot avoid the fact that certain properties are valid everywhere. For this purpose, the logic has the $\Box$ modality: $\Box\varphi$ means that the formula φ is valid everywhere. In the example above, p can access the document doc, if there is a place that has the part doc_2 and can send it everywhere. This can be expressed by the formula $\Diamond(\mathsf{doc}_2 \wedge (\mathsf{doc}_2 \rightarrow \Box\mathsf{doc}_2))$ ***at*** p'. The rules of the logic would allow us to conclude that doc_2 is available at p.

We now define the logic in [10] formally. For the rest of the paper, we shall assume a fixed countable set of atomic formulae *Atoms* and we will vary the set of places. Given a countable set of places *Pl*, let *Frm*(*Pl*) be the set of formulae built from the following grammar:

$$\varphi ::= \top \mid A \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \varphi@p \mid \Box\varphi \mid \Diamond\varphi.$$

Here the syntactic category p stands for elements from *Pl*, and the syntactic category A stands for elements from *Atoms*. The elements in *Frm*(*Pl*) are said *pure formulae*, and are denoted by small Greek letters $\varphi, \psi, \mu \ldots$ An assertion of the form φ ***at*** p is called *sentence*. We denote by capital Greek letters $\Gamma, \Gamma_1, \ldots$ (possibly empty) finite sets of pure formulae, and by capital Greek letters $\Delta, \Delta_1, \ldots$ (possibly empty) finite sets of sentences.

Each judgement in this logic is of the form

$$\Gamma; \Delta \vdash^P \varphi \textbf{ at } p.$$

where

- the *global context* Γ is a (possibly empty) finite set of pure formulae, and represents the properties assumed to hold at every place of the system;
- the *local context* Δ is a (possibly empty) finite set of sentences; since a sentence is a pure formula associated to a place, Δ represents what we assume to be valid in any particular place.
- the sentence φ ***at*** p says that φ is derived to be valid in the place p by assuming $\Gamma; \Delta$.

In the judgement, it is assumed that the places mentioned in Γ and Δ are drawn from the set P. In order to be more formal, we define the function $\mathsf{PL}(X)$, which denotes the set of places that appear in X, for any syntactic object X. It is defined as follow

Definition 1 (Places in a formula). *We define inductively the operator* $\mathsf{PL}(\,)$ *on any syntactic object of the logic as:*

$$\begin{array}{llll}
\mathsf{PL}(A) & \stackrel{\text{def}}{=} \emptyset; & \mathsf{PL}(\top) & \stackrel{\text{def}}{=} \emptyset; \\
\mathsf{PL}(\varphi_1 \wedge \varphi_2) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi_1) \cup \mathsf{PL}(\varphi_2); & \mathsf{PL}(\varphi_1 \to \varphi_2) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi_1) \cup \mathsf{PL}(\varphi_2); \\
\mathsf{PL}(\Box\varphi) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi); & \mathsf{PL}(\Diamond\varphi) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi); \\
\mathsf{PL}(\varphi @ p) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi) \cup \{p\}; & \mathsf{PL}(\varphi \textbf{ at } p) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi) \cup \{p\};
\end{array}$$

$$\begin{array}{ll}
\mathsf{PL}(\varphi_1, \ldots, \varphi_m) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi_1) \cup \ldots \cup \ldots \mathsf{PL}(\varphi_m); \\
\mathsf{PL}(\varphi_1 \textbf{ at } p_1, \ldots, \varphi_n \textbf{ at } p_n) & \stackrel{\text{def}}{=} \mathsf{PL}(\varphi_1 \textbf{ at } p_1) \cup \ldots \cup \mathsf{PL}(\varphi_n \textbf{ at } p_n); \\
\mathsf{PL}(\Gamma; \Delta) & \stackrel{\text{def}}{=} \mathsf{PL}(\Gamma) \cup \mathsf{PL}(\Delta).
\end{array}$$

When we write a judgment of the form $\Gamma; \Delta \vdash^P \varphi$ ***at*** p, then it must be the case that $\mathsf{PL}(\Gamma) \cup \mathsf{PL}(\Delta) \cup \mathsf{PL}(\varphi \textbf{ at } p) \subseteq P$. Any judgment not satisfying this condition is assumed to be undefined.

In Fig. 1 we give the natural deduction for the judgements as defined in [10]. The most interesting of these rules are $\Diamond E$, the elimination of $\Diamond$, and $\Box I$, the introduction of $\Box$. In these rules, we use $P + p$ to denote the disjoint union $P \cup \{p\}$, and witness the fact that the place p does not occur in Γ and Δ. If $p \in P$, then $P + p$, and any judgment containing such notation, is assumed to be undefined in order to avoid a side condition stating this requirement.

The rule $\Diamond E$ explains how we can use the formulae validated at some unspecified location: we introduce a new place and extend the local context by assuming

L

$$\frac{}{\Gamma;\Delta,\varphi\ \mathbf{at}\ p \vdash^P \varphi\ \mathbf{at}\ p}$$

G

$$\frac{}{\Gamma,\varphi;\Delta \vdash^P \varphi\ \mathbf{at}\ p}$$

$\top I$

$$\frac{}{\Gamma;\Delta \vdash^P \top\ \mathbf{at}\ p}$$

$\wedge I$

$$\frac{\Gamma;\Delta \vdash^P \varphi_1\ \mathbf{at}\ p \quad \Gamma;\Delta \vdash^P \varphi_2\ \mathbf{at}\ p}{\Gamma;\Delta \vdash^P \varphi_1 \wedge \varphi_2\ \mathbf{at}\ p}$$

$\wedge E_i$ (i=1,2)

$$\frac{\Gamma;\Delta \vdash^P \varphi_1 \wedge \varphi_2\ \mathbf{at}\ p}{\Gamma;\Delta \vdash^P \varphi_i\ \mathbf{at}\ p}$$

$\rightarrow I$

$$\frac{\Gamma;\Delta,\varphi\ \mathbf{at}\ p \vdash^P \psi\ \mathbf{at}\ p}{\Gamma;\Delta \vdash^P \varphi \rightarrow \psi\ \mathbf{at}\ p}$$

$@I$

$$\frac{\Gamma;\Delta \vdash^P \varphi\ \mathbf{at}\ p}{\Gamma;\Delta \vdash^P \varphi@p\ \mathbf{at}\ p'}$$

$@E$

$$\frac{\Gamma;\Delta \vdash^P \varphi@p\ \mathbf{at}\ p'}{\Gamma;\Delta \vdash^P \varphi\ \mathbf{at}\ p}$$

$\rightarrow E$

$$\frac{\Gamma;\Delta \vdash^P \varphi \rightarrow \psi\ \mathbf{at}\ p \quad \Gamma;\Delta \vdash^P \varphi\ \mathbf{at}\ p}{\Gamma;\Delta \vdash \psi\ \mathbf{at}\ p}$$

$\Diamond I$

$$\frac{\Gamma;\Delta \vdash^P \varphi\ \mathbf{at}\ p}{\Gamma;\Delta \vdash^P \Diamond\varphi\ \mathbf{at}\ p'}$$

$\Diamond E$

$$\frac{\Gamma;\Delta \vdash^P \Diamond\varphi\ \mathbf{at}\ p' \quad \Gamma;\Delta,\varphi\ \mathbf{at}\ q \vdash^{P+q} \psi\ \mathbf{at}\ p''}{\Gamma;\Delta \vdash^P \psi\ \mathbf{at}\ p''}$$

$\Box I$

$$\frac{\Gamma;\Delta \vdash^{P+q} \varphi\ \mathbf{at}\ q}{\Gamma;\Delta \vdash^P \Box\varphi\ \mathbf{at}\ p}$$

$\Box E$

$$\frac{\Gamma;\Delta \vdash^P \Box\varphi\ \mathbf{at}\ p \quad \Gamma,\varphi;\Delta \vdash^P \psi\ \mathbf{at}\ p'}{\Gamma;\Delta \vdash^P \psi\ \mathbf{at}\ p'}$$

FIGURE 1. Natural deduction.

that the formula is validated there. If any assertion that does not mention the new place is validated thus, then it is also validated using the old local context. The rule $\Box I$ says that if a formula is validated in some new place, without any local assumption on that new place, then that formula must be valid everywhere.

The rules $\Diamond I$ and $\Box E$ are reminiscent of the introduction of the existential quantification, and the elimination of universal quantification in first-order intuitionistic logic. This analogy, however has to be taken carefully. For example, if $\Gamma;\Delta \vdash^P \Diamond\psi\ \mathbf{at}\ p$, then we can show using the rules of the logic that $\Gamma;\Delta \vdash^P \Box\Diamond\psi\ \mathbf{at}\ p$.

3 Kripke Semantics

There are a number of semantics for intuitionistic logic and intuitionistic modal logics that allow for a completeness theorem [2, 11, 18]. In this section we concentrate on the semantics introduced by Kripke [12, 20], as it is convenient for applications and fairly simple. This would provide a formalisation of the intuitive concepts introduced in Section 2.

In Kripke semantics for intuitionistic propositional logic, logical assertions are interpreted over Kripke models. The validity of an assertion depends on its behaviour as the truth values of its atoms change from false to true according to a Kripke model. A Kripke model consists of a *partially ordered* set of *Kripke states*, and an *interpretation*, I, that maps atoms into states. The interpretation tells which atoms are true a state. It is required that if an atom is true in a state, then it must remain true in all larger states. Hence, in a larger state more atoms may become true. Consider a logical assertion built from the atoms $A_1, \cdots, A_n$. The assertion is said to be valid in a state if it continues to remain valid in all larger state.

In order to express the full power of the logic introduced in Section 2, we need to enrich the model by introducing places. We achieve this by associating a fixed set of places *Pls* to each Kripke state. The interpretation, I, in our model maps atoms into places in each state. Since we consider atoms to be resources, the map I tells how resources are distributed in a Kripke state. We require that if I maps an atom into a place in a state, then it would map the atom into that place in all larger states. In terms of resources, it means that places in larger states have possibly more resources. The addition of places makes the Kripke model *distributed* in the obvious sense. We are ready to define Kripke model formally.

Definition 2 (Distributed Kripke Model). *A* distributed Kripke model *is a quadruple* $\mathcal{K} = (K, \leq, Pls, I)$, *where*

- K *is a (non empty) set;*
- $\leq$ *is a partial order on* K*;*
- *Pls is a (non empty) set of places;*
- $I : Atoms \rightarrow Pow(K \times Pls)$ *is such that if* $(k, p) \in I(A)$ *then* $(l, p) \in I(A)$ *for all* $l \geq k$.

for $Pow()$ *the powerset operator.*

The set K is the set of Kripke states, whose elements are denoted by $k, l, \ldots$ Relation $\leq$ is the partial order on the states and I is the interpretation of atoms. The definition tells only how resources, i.e. atoms, are distributed in the system.

In order to give semantics to the whole set of formulae $Frm(Pls)$, we need to extend I. The interpretation of a formula depends on its composite parts and if

it is valid in a given state, then it remains valid at the same places in all larger states. For example, the formula $\varphi \wedge \psi$ is valid in a state k at place p, if both φ and ψ are true at place p in all states $l \geq k$.

The introduction of places in the model allows the interpretation of the spatial modalities of the logic. Formula $\varphi@p$ is satisfied at a place in a state k, if it is true at p in all states $l \geq k$; $\Diamond\varphi$ and $\Box\varphi$ are satisfied at a place in state k, if φ is true respectively at some or at every place in all states $l \geq k$.

We extend now the interpretation of atoms to interpretation of formulae, we use induction on the structure of the formulae.

Definition 3 (Kripke Semantics). *For $\mathcal{K} = (K, \leq, Pls, I)$ a distributed Kripke model, the relation $\models$ between couples (k, p) and pure formulae is inductively defined by*

$$
\begin{array}{ll}
(k,p) \models A & \text{iff } (k,p) \in I(A); \\
(k,p) \models \top & \text{for all } (k,p) \in K \times Pls; \\
(k,p) \models \varphi \wedge \psi & \text{iff } (k,p) \models \varphi \text{ and } (k,p) \in I'(\psi); \\
(k,p) \models \varphi \rightarrow \psi & \text{iff } l \geq k \text{ and } (l,p) \models \varphi \text{ imply } (l,p) \models \psi; \\
(k,p) \models \varphi@q & \text{iff } (k,q) \models \varphi; \\
(k,p) \models \Box\varphi & \text{iff } (k,q) \models \varphi \text{ for all } q \in Pls; \\
(k,p) \models \Diamond\varphi & \text{iff there exists } q \in Pls \text{ such that } (q,k) \models \varphi.
\end{array}
$$

We pronounce $(k, p) \models \varphi$ it as (k, p) forces *φ, or (k, p)* satisfies *φ. We write $k \models \varphi$* **at** *p if $(k, p) \models \varphi$.*

Please note that in this extension, except for logical implication, we have not considered larger states in order to interpret a modality or a connective. It turns out that the satisfaction of a formula in a state implies the satisfaction in all larger states.

Lemma 1 (Kripke Monotonicity). *Given $\mathcal{K} = (K, \leq, Pls, I)$ distributed Kripke model, $\models$ preserves the partial order in K, that is for each $p \in Pls$ and each $\varphi \in Frm(Pls)$, if $l \geq k$ then $(k, p) \models \varphi$ implies $(l, p) \models \varphi$.*

Proof: We proceed by induction on the structure of formulae.

Base case. If $\varphi \in Atoms$ or $\varphi = \top$, the lemma holds by Definitions 2 and 3.

Inductive Hypothesis. We consider a formula $\varphi \in Frm(Pls)$. We assume that for every sub-formula φ_i of φ and for every $p \in Pls$: if $l \geq k$ then $(k, p) \models \varphi_i$ implies $(l, p) \models \varphi_i$. We refer to Definition 3. Cases $\varphi = \varphi_1 \wedge \varphi_2$ and $\varphi = \varphi_1 \rightarrow \varphi_2$ are treated as in [20]. Cases $\varphi = \varphi_1@q$, $\varphi = \Box\varphi_1$ and $\varphi = \Diamond\varphi_1$ are similar. We show only the case $\varphi = \varphi_1@q$. Assume $(p, k) \models \varphi_1@q$, then $(q, k) \models \varphi_1$ by definition, hence $(q, l) \models \varphi_1$ for every $l \geq k$ by inductive hypothesis, and so we conclude that $(p, l) \models \varphi_1@q$. ■

Consider now the distributed database described in Section 2. We can express

the same properties that we inferred in Section 2 by using a distributed Kripke model. Fix a Kripke state k. The assumption that the two parts, $\mathsf{doc}_1, \mathsf{doc}_2$, can be combined in p in a state k to give the document doc can be expressed as $(k, p) \models (\mathsf{doc}_1 \wedge \mathsf{doc}_2) \rightarrow \mathsf{doc}$. If the resources doc_1 and doc_2 are assigned to the place p, i.e., $(k, p) \models \mathsf{doc}_1$ and $(k, p) \models \mathsf{doc}_2$, then, since $(k, p) \models \mathsf{doc}_1 \wedge \mathsf{doc}_2$, it follows that $(k, p) \models \mathsf{doc}$.

Let us consider a slightly more complex situation. Suppose that $k \models \Diamond(\mathsf{doc}_2 \wedge (\mathsf{doc}_2 \rightarrow \Box\mathsf{doc}_2))$ ***at*** p'. According to the semantics of $\Diamond$, there is some place r such that $(k, r) \models \mathsf{doc}_2 \wedge (\mathsf{doc}_2 \rightarrow \Box\mathsf{doc}_2)$. The semantics of $\wedge$ tells us that $(k, r) \models \mathsf{doc}_2$ and $(k, r) \models (\mathsf{doc}_2 \rightarrow \Box\mathsf{doc}_2)$. Since $(k, r) \models \mathsf{doc}_2$, we know from the semantics of $\rightarrow$ that $(k, r) \models \Box\mathsf{doc}_2$, and from $\Box$ that $(k, p) \models \mathsf{doc}_2$. Therefore, if doc_1 is placed at p in the state k, then the whole document doc would becomes available at place p in state k.

3.1 Some useful properties

In order to prove soundness of our semantics, we shall need some important properties of the distributed Kripke models. We state and prove those properties in this section.

Lemma 2 says that if we add a new place which duplicates a specific place in all Kripke states, then the set of valid properties does not change. Moreover, the new place mimics the duplicated place. In order to state this lemma, we first prove that duplication gives us a distributed Kripke model.

PROPOSITION 1 (p-DUPLICATED EXTENSION $\mathcal{K}_{q(p)}$). *Let $\mathcal{K} = (K, \leq, Pls, I)$ be a distributed Kripke model. For $p \in Pls$ and $q \notin Pls$ a new place, let $\mathcal{K}_{q(p)} = (K_{q(p)}, \leq_{q(p)}, Pls_{q(p)}, I_{q(p)})$ where*

- *$K_{q(p)}$ is K;*
- *$\leq_{q(p)}$ is $\leq$;*
- *$Pls_{q(p)}$ is $Pls \cup \{q\}$;*
- *$I_{q(p)} : Atoms \longrightarrow Pow(K_{q(p)} \times Pls_{q(p)})$ is defined as*

$$(k, r) \in I_{q(p)}(A) \text{ iff } \begin{cases} (k, r) \in I(A) \ \ (r \in Pls); \\ (k, p) \in I(A) \ (r = q). \end{cases}$$

Then $\mathcal{K}_{q(p)}$ is a distributed Kripke model, and $\mathcal{K}_{q(p)}$ is said to be a p-duplicated extension of $\mathcal{K}$.

Proof: We just need to check that $I_{q(p)}$ satisfies the monotonicity condition on atoms which follows immediately from definition. ■

We show that p-duplicated extension is conservative over all the formulae that do not mention the added place. Moreover, for all such formulae, the new place mimics the duplicated one.

LEMMA 2 ($\mathcal{K}_{q(p)}$ IS CONSERVATIVE). *Let $\mathcal{K} = (K, \leq, Pls, I)$ be a distributed Kripke model and $\mathcal{K}_{q(p)}$ be its p-duplicated extension. Let $\models$ and $\models_{q(p)}$ extend the interpretation of atoms in $\mathcal{K}$ and $\mathcal{K}_{q(p)}$ respectively. For every $k \in K$ and formula $\varphi \in Frm(Pls)$, we have:*

1. *if $r \in Pls$, then $(k, r) \models_{q(p)} \varphi$ if and only if $(k, r) \models \varphi$; and*
2. *if $r = q$, then $(k, q) \models_{q(p)} \varphi$ if and only if $(k, p) \models \varphi$.*

Proof: We prove both of the properties simultaneously by induction on the structure of formulae in $Frm(Pls)$.

Base of induction. The two properties are verified on atoms by the definition of $I_{q(p)}$, and on $\top$ by Definition 3.

Inductive hypothesis. We consider a formula $\varphi \in Frm(Pls)$ and we assume the points hold for each of its sub-formulae φ_i. In particular we assume that:

1. *if $r \in Pls$, then $(k, r) \models_{q(p)} \varphi_i$ if and only if $(k, r) \models \varphi_i$; and*
2. *if $r = q$, then $(k, q) \models_{q(p)} \varphi_i$ if and only if $(k, p) \models \varphi_i$.*

We consider $r \in Pls$ and fix it. We prove only property *1*, as the treatment of point *2* is analogous. Now, we consider several possibilities for ϕ.

Case $\varphi = \varphi_1 \wedge \varphi_2$. The assertion $(k, r) \models_{q(p)} \varphi_1 \wedge \varphi_2$ iff $(k, r) \models_{q(p)} \varphi_1$ and $(k, r) \models_{q(p)} \varphi_2$. By inductive hypothesis, this is equivalent to $(k, r) \models \varphi_1$ and $(k, r) \models \varphi_2$, which is equivalent to $(k, r) \models \varphi_1 \wedge \varphi_2$ by Definition 3.

Case $\varphi = \varphi_1 \rightarrow \varphi_2$. $(k, r) \models_{q(p)} \varphi_1 \rightarrow \varphi_2$ iff $(l, r) \models_{q(p)} \varphi_1$ implies $(l, r) \models_{q(p)} \varphi_2$ for every $l \geq k$. By inductive hypothesis, this is equivalent to $(l, r) \models \varphi_1$ implies $(l, r) \models \varphi_2$ for every $l \geq h$, and this is equivalent to $(k, r) \models \varphi_1 \rightarrow \varphi_2$.

Case $\varphi = \varphi_1 @ s$. $(k, r) \models_{q(p)} \varphi_1 @ s$ iff $(k, s) \models_{q(p)} \varphi_1$. Moreover, we know that $s \in Pls$ as $\varphi_1 @ s \in Frm(Pls)$. By inductive hypothesis $(k, s) \models_{q(p)} \varphi_1$ iff $(k, s) \models \varphi_1$. By definition, $(k, s) \models \varphi_1$ iff $(k, r) \models \varphi_1 @ s$.

Case $\varphi = \Diamond \varphi_1$. Suppose $(k, r) \models_{q(p)} \Diamond \varphi_1$, then there exists $s \in Pls_{q(p)} = Pls \cup \{q\}$ such that $(k, s) \models_{q(p)} \varphi_1$. If $s \in Pls$, then we use inductive hypothesis (property *1*) to obtain $(k, s) \models \varphi_1$, and therefore $(k, r) \models \Diamond \varphi_1$. Otherwise if $s = q$, then we use inductive hypothesis (property 2) *2* to obtain $(k, p) \models \varphi_1$, and therefore $(k, r) \models \Diamond \varphi_1$.

Vice versa, if $(k, r) \models \Diamond \varphi_1$ then there exists $s \in Pls$ such that $(k, s) \models \varphi_1$. Hence by inductive hypothesis (property *1*) $(k, s) \models_{q(p)} \varphi_1$, and we conclude $(k, r) \models_{q(p)} \Diamond \varphi_1$.

Case $\varphi = \Box \varphi_1$. Suppose that $(k, r) \models_{q(p)} \Box \varphi_1$. This means that $(k, s) \models_{q(p)} \varphi_1$ for every $s \in Pls \cup \{q\}$. We can conclude that $(k, r) \models \Box \varphi_1$ by considering every $s \in Pls$ and applying inductive hypothesis (property *1*).

Vice versa, if $(k, r) \models \Box \varphi_1$ then $(k, s) \models \varphi_1$ for every $s \in Pls$. By inductive hypothesis (property *2)* $(k, s) \models_{q(p)} \varphi_1$ for every $s \in Pls$. Also, since $(k, s) \models \varphi_1$ for every $s \in Pls$, we get $(k, p) \models \varphi_1$. Hence by inductive hypothesis (property *2*),

$(k, q) \models_{q(p)} \varphi_1$. We conclude $(k, t) \models_{q(p)} \varphi_1$ for every $t \in Pls_{q(p)}$, which implies $(k, r) \models_{q(p)} \Box\varphi_1$. ■

Another property of distributed Kripke models is the possibility to rename the places in the model. The property says that if we rename a place in the model, then we do not modify the set of valid properties not involving the renamed place. First we prove that the renamed model is still a distributed Kripke model, then we formalize the property in Lemma 3.

Proposition 2 (p-Renaming $\mathcal{K}_{q/p}$). *Given a distributed Kripke model $\mathcal{K} = (K, \leq, Pls, I)$, where $Pls = P + \{p\}$. For a new place $q \notin P$, we define $\mathcal{K}_{q/p} = (K_{q/p}, \leq_{q/p}, Pls_{q/p}, I_{q/p})$ where*

- *$K_{q/p}$ is K;*
- *$\leq_{q/p}$ is $\leq$;*
- *$Pls_{q/p}$ is $P \cup \{q\}$;*
- *$I_{q/p} : Atoms \longrightarrow Pow(K_{q/p} \times Pls_{q/p})$ is defined as*

$$(k, r) \in I_{q/p)}(A) \text{ iff } \begin{cases} (k, r) \in I(A) \ (r \in P); \\ (k, p) \in I(A) \ (r = q). \end{cases}$$

Then $\mathcal{K}_{q/p}$ is a distributed Kripke model, and $\mathcal{K}_{q/p}$ is said to be a p-renaming *of $\mathcal{K}$.*

Proof: As for Proposition 1, We just need to check that $I_{q(p)}$ satisfies the monotonicity condition on atoms, which follows immediately from definition and the monotonicity of I. ■

By mimicking the proof of Lemma 2, we show that $\mathcal{K}_{q/p}$ is conservative with respect to $\mathcal{K}$ and the renamed place behaves like the original one.

Lemma 3 ($\mathcal{K}_{q/p}$ is conservative). *Let $\mathcal{K} = (K, \leq, Pls, I)$ be a distributed Kripke model such that $Pls = P + \{p\}$ and $\mathcal{K}_{q/p}$ be its p-renaming. Let $\models$ and $\models_{q/p}$ extend the interpretation of atoms in $\mathcal{K}$ and $\mathcal{K}_{q/p}$ respectively. For every $k \in K$ and formula $\varphi \in Frm(P)$, we have:*

1. *if $r \in P$, then $(k, r) \models_{q/p} \varphi$ if and only if $(k, r) \models \varphi$; and*
2. *if $r = q$, then $(k, q) \models_{q/p} \varphi$ if and only if $(k, p) \models \varphi$.*

Proof: We proceed as in the proof of Lemma 2, and prove both of the properties simultaneously by induction on the structure of formulae in $Frm(Pls)$.

Base of induction. The properties are verified on atoms and $\top$ by definition.

Inductive hypothesis. As for Lemma 2, we consider a formula $\varphi \in Frm(Pls)$ and we assume that the two properties hold for each of its sub-formulae φ_i. Inductive cases deal with connectives and modalities. Here we consider only the

two most significant cases and prove property *1*. The other cases can be dealt with easily.

Case $\varphi = \Diamond\varphi_1$. Let $r \in P$ and suppose $(k, r) \models {}_{q/p}\Diamond\varphi_1$. Then, by definition there exists $s \in Pls_{q/p} = P \cup \{q\}$ such that $(k, s) \models {}_{q/p}\varphi_1$. If $s \in P$, we use inductive hypothesis (property *1*) to obtain $(k, s) \models \varphi_1$, and in that case $(k, r) \models \Diamond\varphi_1$ by definition. In the case $s = q$, we use inductive hypothesis (property *2*) to obtain $(k, p) \models \varphi_1$ and so $(k, r) \models \Diamond\varphi_1$. The opposite direction is analogous.

Case $\varphi = \Box\varphi_1$. Suppose $(k, r) \models {}_{q/p}\Box\varphi_1$. Then the definition says that $(k, s) \models {}_{q/p}\varphi_1$ for every $s \in P \cup \{q\}$. We get by using inductive hypothesis

- $(k, s) \models \varphi_1$ for every $s \in P$, and
- $(k, p) \models \varphi_1$

We conclude that $(k, t) \models \varphi_1$ for every $t \in P + \{p\}$, and hence $(k, r) \models \Box\varphi_1$. The opposite direction is analogous. ■

3.2 Soundness

We shall now give a semantics of the judgments introduced in §2 using distributed Kripke models. We shall then show that the semantics is both sound and complete. In order to introduce the semantics, we extend the definition of *validity* for pure formulae to sets of pure formula and sets of sentences.

Definition 4 (Forcing Extension). Let $\mathcal{K} = (K, \leq, Pls, I)$ be a distributed Kripke model. Given Γ, a finite set of pure formulae and Δ, a finite set of sentences, such that $\mathsf{PL}(\Gamma; \Delta) \subseteq Pls$, we say that the Kripke state $k \in K$ forces the couple $\Gamma; \Delta$, (and we write $k \models \Gamma; \Delta$) if

1. $(k, p) \models \varphi$ for every $\varphi \in \Gamma$ and $p \in Pls$;
2. $k \models \psi$ ***at*** q for every ψ ***at*** $q \in \Delta$.

A judgment is respected by a distributed Kripke model, if whenever its assumptions are valid in a Kripke state, then its conclusion is also valid in that state. We are now ready to define the satisfaction of a judgement.

Definition 5 (Satisfaction for a Judgment). We say that $\Gamma; \Delta \models {}^{P}\mu$ ***at*** p, and we read it as "$\Gamma; \Delta \vdash^{P} \mu$ **at** *p is valid*", if

- $PL(\Gamma) \cup PL(\Delta) \cup \{p\} \subseteq P$; and
- for every distributed Kripke model $\mathcal{K} = (K, \leq, Pls, I)$ with $P \subseteq Pls$, it is the case that for every $k \in K$, whenever $k \models \Gamma; \Delta$, then $(k, p) \models \mu$ too.

We prove that the semantics is sound for the judgements of the logic. The proof of soundness depends on Lemma 2 and Lemma 3. We need to show that if a judgement is provable in the natural deduction system, then it is also valid.

THEOREM 1 (SOUNDNESS). *If* $\Gamma;\Delta \vdash^P \mu$ **at** p *is derivable in the logic, then it is valid.*

Proof: The proof proceeds by induction on the number n of inference rules applied in the derivation of the judgement $\Gamma;\Delta \vdash^P \mu$ ***at*** p. The most interesting cases are $\Box I$, the introduction of $\Box$, and $\Diamond E$, the elimination of $\Diamond$.

Base Case ($n = 1$). Suppose the judgment is proved by using axiom L, or the axiom G, or the axiom $\top I$. We consider a model $(K, \leq, Pls, I)$ such that $P \subseteq Pls$. We need to show that for every $k \in K$ if $k \models \Gamma;\Delta$ then $(k, p) \models \mu$.

Suppose the derivation consists of just the axiom L, then the assertion μ ***at*** p is in Δ. Hence, by definition, for every $k \in K$ if $k \models \Gamma;\Delta$ then $(k, p) \models \mu$.

If the derivation consists of just the axiom G, then the formula μ is in Γ, and so $k \models \Gamma;\Delta$ implies $(k, r) \models \mu$ for every $r \in Pls$. In particular $(k, p) \models \mu$.

Finally If the derivation is the application of $\top I$, then μ is $\top$ and the result holds by definition.

Inductive hypothesis ($n > 1$). We assume the theorem holds for any judgment that is deducible by applying less than n instances of inference rules. In particular we assume that:
If the judgment $\Gamma;\Delta \vdash^P \mu$ **at** p *is deducible in the logic by using less than n instances of the rules, then* $\Gamma;\Delta \models {}^P\mu$ **at** p.

We consider a judgment $\Gamma;\Delta \vdash^P \mu$ ***at*** p which is derivable in the logic by using exactly n instances of inference rules. We fix a model $\mathcal{K} = (K, \leq, Pls, I)$ such that $P \subseteq Pls$, and let $\models$ be the extension of I on $Frm(Pls)$. We fix $k \in K$ such that $k \models \Gamma;\Delta$. We need to prove $(k, p) \models \mu$. We consider the last rule applied to obtain $\Gamma;\Delta \vdash^P \mu$ ***at*** p, and proceed by cases. In most cases, we apply the inductive hypothesis on the model $\mathcal{K}$ only. However, for $\Box I$ and $\Diamond E$ we will use inductive hypothesis on an extension of $\mathcal{K}$.

Cases $\wedge I$ and $\wedge E$ follow from Definition 3 and are treated as in [20].

Case $\rightarrow I$. Then $\mu = \varphi \rightarrow \psi$ and we can derive $\Gamma;\Delta, \varphi$ ***at*** $p \vdash^P \psi$ ***at*** p by applying $n - 1$ instances of the rules. The inductive hypothesis says that for every $l \in K$: $l \models \Gamma;\Delta, \varphi$ ***at*** p implies $l \models \psi$ ***at*** p.

Let $l \geq k$. Then $l \models \Gamma;\Delta$ by Kripke Monotonicity (Lemma 1). If we assume $(l, p) \models \varphi$, then the inductive hypothesis says that $(l, p) \models \psi$ too. Hence, we have that for all $l \geq k$, if $l \models \varphi$ then $l \models \psi$ also. We conclude that $(k, p) \models \varphi \rightarrow \psi$ by definition of $\models$.

Case $\rightarrow E$. Then, we have that $\Gamma;\Delta \vdash^P \varphi \rightarrow \mu$ and $\Gamma, \Delta \vdash^P \varphi$ for some φ. The inductive hypothesis says that $(k, p) \models \varphi \rightarrow \mu$ and $(k, p) \models \varphi$. Hence, we get $(k, p) \models \mu$ according to Definition 3.

Case $@I$. Then μ is of the form $\varphi @ q$, and $\Gamma;\Delta \vdash^P \varphi$ ***at*** q. The inductive hypothesis says that $(k, q) \models \varphi$, and hence $(k, p) \models \varphi @ q$.

Case $@E$. Then we have that $\Gamma;\Delta \vdash^P \mu @ p$ ***at*** q for some $q \in P$. The

inductive hypothesis says that $(k, q) \models \varphi@p$, and therefore $(k, p) \models \varphi$.

Case $\Box I$. Then μ is of the form $\Box\varphi$. Moreover $\Gamma; \Delta \vdash^{P+q} \varphi$ ***at*** p_1 for some $p_1 \notin P$ by using $n-1$ instances of the inference rules. By inductive hypothesis we know that $\Gamma; \Delta \models^{P+p_1} \varphi$ ***at*** q. Please note that since $\Gamma; \Delta \vdash^{P} \mu atp$, we also have $\mathsf{PL}(\Gamma; \Delta) \cup \mathsf{PL}(\varphi) \subseteq P$. Let *Pls* be $P + p_1$.

First, consider the case when $p_1 \notin Pls$. We need to show that $k \models \Box\varphi$ ***at*** p. According to semantics of $\Box$, it suffices to show that $k \models \varphi$ ***at*** r, for all $r \in Pls$. Fix one $r \in Pls$, and consider the r-duplicated extension $\mathcal{K}_{q(r)}$. Let $\models_{q(r)}$ be the extension of $I_{q(r)}$. We get $k \models_{q(r)} \Gamma; \Delta$ by using Lemma 2 (since $k \models \Gamma; \Delta$).

Now, we have that $\Gamma; \Delta \vdash^{P+q} \varphi$ ***at*** p_1 .and $P + p_1 \subseteq Pls_{q(r)}$. Since $k \models_{q(r)} \Gamma; \Delta$, we get by using inductive hypothesis on $\mathcal{K}_{q(r)}$ that $(k, p_1) \models_{q(r)} \varphi$. Now, we can conclude $(k, r) \models \varphi$ ***at*** r by using Lemma 2.

Since r was arbitrary, we deduce $k \models \Box\varphi$ ***at*** p.

If $p_1 \in Pls$, then $Pls = Pls' + \{p_1\}$ with $\mathsf{PL}(\Gamma; \Delta) \cup \mathsf{PL}(\varphi) \subseteq P \subseteq Pls'$. We choose $t \notin Pls$ and consider $\mathcal{K}_{t/p_1}$ to be the p_1-renaming of $\mathcal{K}$, as defined in Proposition 2. Let $\models_{t/p_1}$ be the extension of I_{t/p_1}. By following the above reasoning we derive $k \models_{t/p_1} \Box\varphi$ ***at*** p, hence $k \models \Box\varphi$ ***at*** p by Lemma 3.

Case $\Box E$. Then we have that there is some formula φ such that $\Gamma; \Delta \vdash^{P} \Box\varphi$ ***at*** p_1 and $\Gamma, \varphi; \Delta \vdash^{P} \Box\mu$ ***at*** p by using less than n instances of inference rules. The inductive hypothesis on $\Gamma; \Delta \vdash^{P} \Box\varphi$ ***at*** p implies $(k, p_1) \models \Box\varphi$, and this means that $(k, q) \models \varphi$ for every $q \in Pls$. By definition, we obtain $k \models \Gamma, \varphi; \Delta$ and using inductive hypothesis on $\Gamma, \varphi; \Delta \vdash^{P} \Box\mu$ ***at*** p we conclude $(k, p') \models \psi$.

Case $\Diamond I$. Then we have that μ is of the form $\Diamond\varphi$ for some formula φ, and $\Gamma; \Delta \vdash^{P} \varphi$ ***at*** p_1 for some p_1. The inductive hypothesis says that $(k, p_1) \models \varphi$, so we conclude $(k, p) \models \Diamond\varphi$.

Case $\Diamond E$. Then for some $p' \in P$ and $\varphi \in Frm(P)$ we can derive $\Gamma; \Delta \vdash^{P} \Diamond\varphi$ ***at*** p' and $\Gamma; \Delta, \varphi$ ***at*** $q \vdash^{P+q} \mu$ ***at*** p by using less than n instances of the rules. Hence by inductive hypothesis: $\Gamma; \Delta \models^{P} \Diamond\varphi$ ***at*** p' and $\Gamma; \Delta, \varphi$ ***at*** $q \models^{P+q} \mu$ ***at*** p.

As in the case for $\Box I$, first assume $q \notin Pls$. We need to show that $(k, p) \models \mu$. Since $k \models \Gamma; \Delta$ we get $(k, p') \models \Diamond\varphi$, and this means that there exists $r \in Pls$ such that $(k, r) \models \varphi$.

Consider now the r-duplicated extension $\mathcal{K}_{q(r)}$ of $\mathcal{K}$. Let $\models_{q(r)}$ be the extension of $I_{q(r)}$. By Lemma 2 we have $(k, q) \models_{q(r)} \varphi$, and $k \models_{q(r)} \Gamma; \Delta$. Hence, we get $k \models_{q(r)} \Gamma; \Delta, \varphi$ ***at*** q. Since $\Gamma; \Delta, \varphi$ ***at*** $q \models^{P+q} \mu$ ***at*** p, we get $(k, p) \models_{q(r)} \mu$. As $\mathsf{PL}(\mu) \subseteq P \subseteq Pls$ and $p \in P \subseteq Pls$, we obtain $(k, p) \models \mu$ by Lemma 2.

In the case that $\mathcal{K}$ is such that $q \in Pls$, we can rename q by a fresh as we did in $\Box I$, and obtain the desired result. ■

3.3 Completeness

We shall show that our semantics is complete for the natural deduction in Section 2. First, we extend the notion of provability to possible non-finite sets Σ of

sentences by saying that $\Gamma; \Sigma \vdash^P \varphi$ ***at*** q, if and only if, there exists a finite set $\Delta \subseteq \Sigma$ such that $\Gamma; \Delta \vdash^P \varphi$ ***at*** q.

As in standard proofs of completeness of intuitionistic logics[20, 18, 2], the proof of completeness is based on the construction of a particular distributed Kripke model: the *canonical model*. We will prove that a sequent is valid in the canonical model if and only if it is derivable in the logic. In the construction of the canonical model, we consider particular kinds of sets of formulae.

Definition 6 (Prime Set). *Given a set of places Pls and a finite set* Γ *of pure formulae in Frm(Pls), a (possibly non-finite) set* Σ *of sentences with* $\mathsf{PL}(\Sigma) \subseteq Pls$*, is said to be* (Γ, Pls)-prime *if for every formula* $\varphi \in Frm(Pls)$*:*

1. $\Gamma; \Sigma \vdash^{Pls} \Diamond\varphi$ **at** p*, implies that there exists* $q \in Pls$ *s.t.* $\Gamma; \Sigma \vdash^{Pls} \varphi$ **at** q*;*
2. $\Gamma; \Sigma \vdash^{Pls} \varphi$ **at** r *for all* $r \in Pls$*, implies* $\Gamma; \Sigma \vdash^{Pls} \Box\varphi$ **at** p *for all* $p \in Pls$*.*

The canonical model will be built by choosing the prime sets of formulae as Kripke states. We would show that given Γ and Δ, we can construct a set of places *Pls* and a prime set $\Sigma \supseteq \Delta$ such that Σ is (Γ, Pls)-prime. Before we proceed, we first state a proposition proved in [9]:

Proposition 3. *Let* $P \subseteq P'$ *and suppose* $\mathsf{PL}(\Gamma) \cup \mathsf{PL}(\Delta) \cup \mathsf{PL}(\varphi$ **at** $p) \subseteq P$*, then* $\Gamma; \Delta \vdash^{P'} \varphi$ **at** p *if and only if* $\Gamma; \Delta \vdash^P \varphi$ **at** p *.*

Now, we show the existence of prime extensions:

Lemma 4 (Prime Extension). *Let P be a set of places and* Γ *be a finite set of pure formulae in Frm(P). For every finite set* Δ *of sentences such that* $\mathsf{PL}(\Delta) \subseteq P$*, there exists a set of places* P' *extending P and a* (Γ, P')*-prime set of sentences* Σ *containing* Δ*, such that given* $\varphi \in Frm(P)$ *and* $p \in P$*:*

$$\Gamma; \Delta \vdash^P \varphi \textbf{ at } p \textit{ if and only if } \Gamma; \Sigma \vdash^{P'} \varphi \textbf{ at } p.$$

Proof: We enrich the set of places by introducing two kind of places: $\mathbf{q}_i$, which will be the witnesses for the formulae $\Diamond\varphi$, and $\mathbf{p}_j$, which will be the new places used to introduce $\Box\psi$ in the case ψ is provable for every place.

The set of places P' is obtained by a series of extensions $P = P_0 \subseteq P_1 \subseteq P_2 \ldots$ The sets P_{n+1} are constructed as $P_{n+1} = P_n \cup \{\mathbf{q}_{n+1}, \mathbf{p}_{n+1}\}$, where the places $\mathbf{q}_{n+1}, \mathbf{p}_{n+1}$ are new, i.e., $\mathbf{q}_{n+1}, \mathbf{p}_{n+1} \notin P_n$. Also, $\mathbf{q}_{n+1}$ is different from $\mathbf{p}_{n+1}$. The set P' is taken as $P' = \bigcup_{n \geq 0} P_n$.

Before we proceed with the construction, we pick up an enumeration of the pure formulae $Frm(P')$, and fix it. The set Σ is obtained by series of extensions $\Delta = \Sigma_0 \subseteq \Sigma_1 \subseteq \Sigma_2 \ldots$ that verify the following:

Property 1. For every $n \geq 0$*:*

1. $\mathsf{PL}(\Sigma_n) \subseteq P_n$.

2. *Given* $\varphi \in Frm(P_n)$ *and* $p \in P_n$, *we have* $\Gamma; \Delta \vdash^{P} \varphi$ **at** p *if and only if* $\Gamma; \Sigma_n \vdash^{P_n} \varphi$ **at** p.

The series is constructed inductively. In the induction, we will create witnesses for the formulae of the type $\Diamond\psi$. We shall also construct a set, $treated_n$, of formulae of the sort $\Diamond\psi$. This set, initialised to be the empty set, will be the set of the formulae for which we have already created witnesses.

We put $treated_0 = \emptyset$, $P_0 = P$ and $\Sigma_0 = \Delta$. It is clear that $\mathsf{PL}(\Sigma_0) \subseteq P_0$, and $\Gamma; \Delta \vdash^{P} \varphi$ ***at*** p if and only if $\Gamma; \Sigma_0 \vdash^{P_0} \varphi$ ***at*** p.

Now, we proceed inductively. Let Σ_n ($n \geq 0$) extend Δ and satisfying Property 1. In step $n + 1$, we pick the first formula $\Diamond\psi$ in the enumeration such that

- $\Diamond\psi$ is in $Frm(P_n)$, i.e., all the places in $\Diamond\psi$ are taken from P_n;
- $\Diamond\psi \notin treated_n$; and
- $\Gamma; \Sigma_n \vdash^{P_n} \Diamond\psi$ ***at*** q, for some $q \in P_n$.

We define $\Sigma_{n+1} = \Sigma_n \cup \{\psi$ ***at*** $\mathbf{q}_{n+1}\}$ and $treated_{n+1} = treated_n \cup \{\Diamond\psi\}$. The place $\mathbf{q}_{n+1}$ witnesses the existential $\Diamond$. Clearly $\mathsf{PL}(\Sigma_{n+1}) \subseteq P_{n+1}$. Now we prove the following:

Claim. For any $\varphi \in Frm(P_n)$ and $p \in P_n$, $\Gamma; \Sigma_n \vdash^{P_n} \varphi$ ***at*** p if and only if $\Gamma; \Sigma_{n+1} \vdash^{P_{n+1}} \varphi$ ***at*** p.

The direction from left to right is a consequence of inference rule L, and Proposition 3. In order to prove the converse, assume $\Gamma; \Sigma_{n+1} \vdash^{P_{n+1}} \varphi$ ***at*** p. Now let ψ be the formula chosen at step $n + 1$. We have by construction, $\Gamma; \Sigma_n \vdash^{P_n} \Diamond\psi$ ***at*** q. Also since $\Gamma; \Sigma_{n+1} \vdash^{P_{n+1}} \varphi$ ***at*** p, we get by using the inference rule L and Proposition 3 that $\Gamma; \Sigma_n, \psi$ ***at*** $\mathbf{q}_n \vdash^{P_n + \mathbf{q}_{n+1}} \varphi$ ***at*** p. Hence, we get $\Gamma; \Sigma_n \vdash^{P_n} \varphi$ ***at*** p by application of the inference rule $\Diamond E$.

Suppose now that $\varphi \in Frm(P)$ and $p \in P$. We can assert using the claim above that $\Gamma; \Delta \vdash^{P} \varphi$ ***at*** p if and only if $\Gamma; \Sigma_{n+1} \vdash^{P_{n+1}} \varphi$ ***at*** p. We have just proved Property 1 for the inductive step n.

Finally, we define $\Sigma = \bigcup_{n \geq 0} \Sigma_n$. Clearly $\Gamma; \Delta \vdash^{P} \varphi$ ***at*** p implies $\Gamma; \Sigma \vdash^{P'} \varphi$ ***at*** p, by definition and Proposition 3.

In the other direction, suppose $\Gamma; \Sigma \vdash^{P'} \varphi$ ***at*** p with $\varphi \in Frm(P)$ and $p \in P$. According to the definition, there exists a finite sequence $\Lambda \subseteq \Sigma$ such that $\Gamma; \Lambda \vdash^{P'} \varphi$ ***at*** p. We can then choose $n \geq 0$ big enough to have $\Lambda \subseteq \Sigma_n$ and so $\Gamma; \Sigma_n \vdash^{P'} \varphi$ ***at*** p by the inference rule L. Using Proposition 3 once again, we have $\Gamma; \Sigma_n \vdash^{P_n} \varphi$ ***at*** p. Since $\mathsf{PL}(\Gamma), \mathsf{PL}(\varphi), \{p\} \subseteq P \subseteq P_n$, we conclude $\Gamma; \Delta \vdash^{P} \varphi$ ***at*** p using Proposition 1.

All we need to prove now is that Σ is (Γ, P')-prime.

1. If $\Gamma; \Sigma \vdash^{P'} \Diamond\varphi$ ***at*** p, let n be the least such that $\Diamond\varphi \in \mathsf{PL}(P_n)$ and $p \in P_n$. By construction, there is some $m \geq n$, such that $\Diamond\varphi$ is picked in the construction

of Σ_m. Hence φ ***at*** $\mathbf{q}_m \in \Sigma_m \subseteq \Sigma$, and we conclude that $\Gamma; \Sigma \vdash^{P'} \varphi$ ***at*** $\mathbf{q}_m$.

2. Let $\psi \in \mathit{Frm}(P')$ and suppose $\Gamma; \Sigma \vdash^{P'} \psi$ ***at*** p for all $p \in P'$. In particular, consider the place $\mathbf{p}_n$, with n such that $\psi \in \mathit{Frm}(P_n)$. We have that $\Gamma; \Sigma \vdash^{P'} \psi$ ***at*** $\mathbf{p}_n$.

   Using Proposition 3, we can find $m \geq 0$ such that $\Gamma; \Sigma_m \vdash^{P_m} \psi$ ***at*** $\mathbf{p}_n$. If $m > n$ then we use the above claim iteratively to conclude $\Gamma; \Sigma_n \vdash^{P_n} \psi$ ***at*** $\mathbf{p}_n$. In the case $m \leq n$ we obtain the same conclusion by the inference rule *L*.

   Since $\mathbf{p}_n \notin \mathsf{PL}(\Sigma_n)$ by construction, we can infer that $\Gamma; \Sigma_n \vdash^{P_n \setminus \{\mathbf{p}_n\}} \Box\psi$ ***at*** p for all $p \in P_n$ by the inference rule $\Box I$. Hence $\Gamma; \Sigma \vdash^{P'} \Box\psi$ ***at*** p for all $p \in P_n \setminus \{\mathbf{p}_n\}$ by Proposition 3.

   We conclude by extending $\Gamma; \Sigma \vdash^{P'} \Box\psi$ ***at*** r to any $r \in (P' \setminus P_n) \cup \{\mathbf{p}_n\}$ in the following way (here z' is chosen to be a place $\notin P'$):

$$\dfrac{\Gamma; \Sigma \vdash^{P'} \Box\psi \textbf{ at } p \qquad \dfrac{\dfrac{}{\Gamma, \psi; \Sigma \vdash^{P'+z} \psi \textbf{ at } z}\, G}{\Gamma, \psi; \Sigma \vdash^{P'} \Box\psi \textbf{ at } r}\, \Box I}{\Gamma; \Sigma \vdash^{P'} \Box\psi \textbf{ at } r}\, \Box E$$

■

We are ready to define the canonical model for a finite set of pure formulae Γ and places *Pls*. In this model, the worlds will be (Γ, Pls)-prime sets. The partial order will be subset inclusion, and the atoms will be placed in a specific place p in a world Σ if $\Gamma; \Sigma \vdash^{Pls} A$ ***at*** p.

Definition 7 (Canonical Model). *Given a set of places Pls and a finite set* Γ *of pure formulae in* $\mathit{Frm}(\mathit{Pls})$*, we define the* (Γ, Pls)-canonical *to be the quadruple* $\mathcal{M}_{(\Gamma,\mathit{Pls})} = (M, \subseteq, \mathit{Pls}, I_\Gamma)$*, where:*

- *M is composed by all the* (Γ, Pls)*-prime sets;*
- $\subseteq$ *is set inclusion;*
- $I_\Gamma : \mathit{Atoms} \longrightarrow \mathit{Pow}(M \times \mathit{Pls})$ *is defined by:* $(\Sigma, p) \in I_\Gamma(A)$ *iff* $\Gamma; \Sigma \vdash^{Pls} A$ **at** p.

We now show that the model is a distributed Kripke model. We will also demonstrate that the extension of I_Γ to interpretation of formulae corresponds exactly to the provability in the logic, i.e., $(\Sigma, q) \models \psi$ in the canonical model if and only if $\Gamma; \Sigma \vdash^{Pls} \psi$ ***at*** q.

Lemma 5 (Canonical Evaluation). *Given a set of places Pls and a finite set* Γ *of pure formulae in* $\mathit{Frm}(\mathit{Pls})$*, we have:*

1. *the* (Γ, Pls)*-canonical model* $\mathcal{M}_{(\Gamma,\mathit{Pls})} = (M, \subseteq, \mathit{Pls}, I_\Gamma)$ *is a distributed Kripke model;*

2. *for all* $\varphi \in Frm(Pls)$, $\Sigma \in M$ *and* $q \in Pls$: $(\Sigma, q) \models \varphi$ *if and only if* $\Gamma; \Sigma \vdash^{Pls} \varphi$ **at** q.

Proof: Clearly the inclusion among sets $\subseteq$ is a partial order on M and I_Γ is monotone on M, since if $\Sigma_1 \subseteq \Sigma_2$ then $\Gamma; \Sigma_1 \vdash^{Pls} A$ ***at*** p implies $\Gamma; \Sigma_2 \vdash^{Pls} A$ ***at*** p by definition. All we have to prove is the part *2* of the proposition. We proceed by induction on the structure of the formula φ and we prove that for every $\Sigma \in M$ and $q \in Pls$: $(\Sigma, q) \models \varphi$ if and only if $\Gamma; \Sigma \vdash^{Pls} \varphi$ ***at*** q.

Base Case. The property is verified on *Atoms*, by the definition of I_Γ, and on $\top$, by Definition 3.

Inductive hypothesis. We assume the property holds for any sub-formula of the formula φ we are considering. In particular we assume that:
Given φ_i *sub-formula of* $\varphi \in Frm(Pls)$*, then for every* $\Sigma \in M$ *and* $q \in Pls$*:* $(\Sigma, q) \models \varphi_i$ *if and only if* $\Gamma; \Sigma \vdash^{Pls} \varphi_i$ **at** q.

We need to show that $(\Sigma, q) \models \varphi$ if and only if $\Gamma; \Sigma \vdash^{Pls} \varphi$ ***at*** q. We proceed by cases on structure of φ. The cases in which φ is $\varphi_1 \wedge \varphi_2$, and φ is $\varphi_1 \rightarrow \varphi_2$ are fairly standard. We just consider the three modalities.

Case $\varphi_1 @ p$. Suppose that $(\Sigma, q) \models \varphi_1 @ p$. By definition, we have $(\Sigma, p) \models \varphi_1$. We get $\Gamma; \Sigma \vdash^{Pls} \varphi_1$ ***at*** p by inductive hypothesis. We can conclude $\Gamma; \Sigma \vdash^{Pls} \varphi_1 @ p$ ***at*** q by using the inference rule $@I$.

In the other direction, the fact $\Gamma; \Sigma \vdash^{Pls} \varphi_1 @ p$ ***at*** q implies $\Gamma; \Sigma \vdash^{Pls} \varphi_1$ ***at*** p by using the inference rule $@E$. Hence $(\Sigma, p) \models \varphi_1$ by inductive hypothesis, and therefore $(\Sigma, q) \models \varphi_1 @ p$.

Case $\Box\varphi_1$. $(\Sigma, q) \models \Box\varphi_1$ implies $(\Sigma, p) \models \varphi_1$ for all $p \in Pls$. By inductive hypothesis, this is $\Gamma; \Sigma \vdash^{Pls} \varphi_1$ ***at*** p for all $p \in Pls$. Since Σ is (Γ, Pls) prime, we can conclude $\Gamma; \Sigma \vdash^{Pls} \Box\varphi_1$ ***at*** q

In the other direction, let us assume that $\Gamma; \Sigma \vdash^{Pls} \Box\varphi_1$ ***at*** q. We apply the inference rule $\Box E$ to obtain $\Gamma; \Sigma \vdash^{Pls} \varphi_1$ ***at*** p for every $p \in Pls$. Hence $(\Sigma, p) \models \varphi_1$ for every $p \in Pls$, and therefore $(\Sigma, q) \models \Box\varphi_1$.

Case $\Diamond\varphi_1$. $(\Sigma, q) \models \Diamond_1\varphi$ says that there exists $p \in Pls$ such that $(\Sigma, p) \models \varphi_1$. Using inductive hypothesis, we get $\Gamma; \Sigma \vdash^{Pls} \varphi_1$ ***at*** p. We conclude $\Gamma; \Sigma \vdash^{Pls} \Diamond\varphi_1$ ***at*** q by $\Diamond I$.

In the other direction, assume $\Gamma; \Sigma \vdash^{Pls} \Diamond\varphi_1$ ***at*** q. Since Σ is (Γ, Pls) prime, there exists $p \in Pls$ such that $\Gamma; \Sigma \vdash^{Pls} \varphi_1$ ***at*** p. Using inductive hypothesis, we obtain $(\Sigma, p) \models \varphi_1$. We get $(\Sigma, q) \models \Diamond\varphi_1$ according to Definition 3. ■

Finally we use the canonical model to prove completeness.

THEOREM 2 (COMPLETENESS). $\Gamma; \Delta \models {}^{P}\mu$ **at** $p \Longrightarrow \Gamma; \Delta \vdash^{P} \mu$ **at** p.

Proof: Assume $\Gamma; \Delta \models {}^{P}\varphi$ ***at*** p. This means that

- $PL(\Gamma) \cup PL(\Delta) \cup \{p\} \subseteq P$; and
- for every distributed Kripke model $\mathcal{K} = (K, \leq, Pl, I)$ with $P \subseteq Pl$, it is the

case that for every $k \in K$, whenever $k \models \Gamma; \Delta$ then $(k, p) \models \mu$ also.

We need to show that $\Gamma; \Delta \vdash^P \varphi$ ***at*** p.

Using Lemma 4, construct a set of places $Pls \supseteq P$, and a (Γ, Pls)-prime set of sentence Σ such that: *for every* $\varphi \in Frm(P)$ *and* $p \in P$ $\Gamma; \Delta \vdash^P \varphi$ **at** p *if and only if* $\Gamma; \Sigma \vdash^{Pls} \varphi$ **at** p.

Consider now the (Γ, Pls)-canonical model, as stated in Definition 7. In the canonical model, the worlds are the (Γ, Pls)-prime sets and the set of places is Pls. We focus our attention on the world Σ.

First we claim that in the canonical model $\Sigma \models \Gamma; \Delta$. In order to show this, we need the following:

- For every $\psi \in \Gamma, q \in Pls$, we need to show that $\Sigma \models \psi$ ***at*** q. Given $\psi \in \Gamma$, an application of inference rule G (see figure 1) gives us $\Gamma; \Sigma \vdash^{Pls} \psi$ ***at*** q. By Lemma 5, $\Sigma \models \psi$ ***at*** q if and only if $\Gamma; \Sigma \vdash^{Pls} \psi$ ***at*** q. Hence, we get $\Sigma \models \psi$ ***at*** q.
- For every ψ ***at*** $q \in \Delta$, we need to show that $\Sigma \models \psi$ ***at*** q. Given ψ ***at*** $q \in \Delta$, an application of the L rule (see figure 1), gives us that $\Gamma; \Delta \vdash^{Pls} \psi$ ***at*** q. Σ extends Δ, and hence we get $\Gamma; \Sigma \vdash^{Pls} \psi$ ***at*** q. By Lemma 5 once again, we get $\Sigma \models \psi$ ***at*** q.

So we have a model in which $\Sigma \models \Gamma; \Delta$. By assumption, this implies $\Sigma \models \mu$ ***at*** p. Using Lemma 5, we get that $\Gamma, \Sigma \vdash^{Pls} \mu$ ***at*** p. Since Σ is a prime extension of Δ constructed through Lemma 4, we conclude $\Gamma; \Delta \vdash^P \mu$ ***at*** p. ■

4 Hybrid IS5

We now extend the logic in [9, 10] with disjunctive connectives, thus achieving the full set of intuitionistic connectives. Given a set of places, Pl, the new set of pure formulae (see section 2), $Frm(Pl)$, is the set of formulae built from the following grammar:

$$\varphi ::= \top \mid \bot \mid A \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi @ p \mid \Box\varphi \mid \Diamond\varphi.$$

To account for the new connectives, we extend the natural deduction presented in Figure 1 with rules for the disjunctive connectives. These rules are given in Figure 2. Please note that the rule $\bot E$ as stated has a local flavour: from $\bot$ ***at*** p, we can infer any other property in the same place, p. However, the rule has a "global" consequence. If we have $\bot$ ***at*** p, then we can infer $\bot @ q$ ***at*** p. Using $@E$, we can then infer $\bot$ ***at*** q. Hence if a set of assumptions make a place to be inconsistent, then it will make all places to be inconsistent.

As we shall see in section 5, the Kripke semantics of this extended logic would be similar to the one given for intuitionistic system *S5* [18]. Hence this logic can be seen as an instance of *Hybrid IS5* [2].

$$\frac{}{\Gamma;\Delta \vdash^P \bot \ \textbf{\textit{at}}\ p}\ \bot E$$

$$\frac{\Gamma;\Delta \vdash^P \varphi_i \ \textbf{\textit{at}}\ p}{\Gamma;\Delta \vdash^P \varphi_1 \vee \varphi_2 \ \textbf{\textit{at}}\ p}\ \vee I\ (i{=}1,2)$$

$$\frac{\Gamma;\Delta \vdash^P \varphi_1 \vee \varphi_2 \ \textbf{\textit{at}}\ p \quad \Gamma;\Delta,\varphi_1 \ \textbf{\textit{at}}\ p \vdash^P \psi \ \textbf{\textit{at}}\ p \quad \Gamma;\Delta,\varphi_2 \ \textbf{\textit{at}}\ p \vdash^P \psi \ \textbf{\textit{at}}\ p}{\Gamma;\Delta \vdash^P \psi \ \textbf{\textit{at}}\ p}\ \vee E$$

FIGURE 2. Disjunctive rules

5 Refined Kripke Semantics

We were unable to prove completeness for the extended logic using the semantics defined in Section 3. We had to change the semantics in order to obtain a completeness result, and we present the semantics in this section. The difference from the model of Section 3, is that the set of places in Kripke states are not fixed and may vary. However, they change in a conservative way in that the set of places in a Kripke state is always contained in larger Kripke states. We now present the extended Kripke models which we shall call *Refined Distributed Kripke models*.

DEFINITION 8 (REFINED DISTRIBUTED KRIPKE MODEL). *A quadruple* $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ *is called* refined distributed Kripke model *if*

- K *is a (non empty) set;*
- $\leq$ *is a partial order on* K*;*
- P_k *is a* non-empty *set of places for all* $k \in K$*;*
- $P_k \subseteq P_l$ *if* $k \leq l$*;*
- $I_k : Atoms \rightarrow Pow(P_k)$ *is such that if* $p \in I_k(A)$ *then* $p \in I_l(A)$ *for all* $l \geq k$*.*

Let $Pls = \cup_{k\in K} P_k$*. We shall say that* Pls *is the set of places of* $\mathcal{K}_{ref}$*.*

We extend the forcing relation of Def. 3. The difference from that relation is that the interpretation for $\Box$ changes. This is because larger Kripke states may have more places. Hence when interpreting $\Box\phi$ at a place in particular Kripke state, we have to account for places that may exist in a larger Kripke state. If we stick to the old interpretation, then Kripke monotonicity would fail. The interpretation of $\Box$ is similar to those used for modal intuitionistic logic [2, 18].

DEFINITION 9 (REFINED SEMANTICS). *Let* $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ *be a re-*

fined distributed Kripke model with set of places, Pls. Given $k \in K$, $p \in P_k$, *a pure formula* φ *with* $\mathsf{PL}(\varphi) \subseteq Pls$, *we define* $(k, p) \models \varphi$ *inductively as:*

$$
\begin{array}{ll}
(k,p) \models A & \textit{iff } p \in I_k(A); \\
(k,p) \models \top & \textit{iff } p \in P_k; \\
(k,p) \models \bot & \textit{never;} \\
(k,p) \models \varphi \wedge \psi & \textit{iff } (k,p) \models \varphi \textit{ and } (k,p) \models \psi; \\
(k,p) \models \varphi \vee \psi & \textit{iff } (k,p) \models \varphi \textit{ or } (k,p) \models \psi; \\
(k,p) \models \varphi \to \psi & \textit{iff } l \geq k \textit{ and } (l,p) \models \varphi \textit{ imply } (l,p) \models \psi; \\
(k,p) \models \varphi@q & \textit{iff } q \in P_k \textit{ and } (k,q) \models \varphi; \\
(k,p) \models \Box\varphi & \textit{iff } l \geq k \textit{ and } q \in P_l \textit{ imply } (l,q) \models \psi; \\
(k,p) \models \Diamond\varphi & \textit{iff there exists } q \in P_k \textit{ such that } (q,k) \models \varphi.
\end{array}
$$

We pronounce $(k, p) \models \varphi$ *as* (k, p) ref-forces φ, *or* (k, p) ref-satisfies φ. *We write* $k \models \varphi$ **at** p *if* $(k, p) \models \varphi$.

It is clear from the definition that if $k \models \varphi$ ***at*** p, then $\mathsf{PL}(\varphi$ ***at*** $p) \subseteq P_k$. Moreover, the usual Kripke monotonicity still holds.

Lemma 6 (Kripke Monotonicity). *Let* $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ *be a refined distributed Kripke model with set of places, Pls. The relation* $\models$ *preserves the partial order on* K, i.e., *for each* $k, l \in K$, $p \in P_k$, *and* $\varphi \in Frm(P_k)$, *if* $l \geq k$ *then* $(k, p) \models \varphi$ *implies* $(l, p) \models \varphi$.

Proof: By induction on the structure of formulae, and is similar to the proof for Lemma1. ■

Now, we are ready to extend the definition of forcing to judgements. First, we extend the definition to contexts.

Definition 10 (Forcing on Contexts). *Let* $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ *be a refined distributed Kripke model. Given* $k \in K$, *a finite set of pure formulae* Γ, *and a finite set of sentences* Δ, *such that* $\mathsf{PL}(\Gamma; \Delta) \subseteq P_k$, *we say that* k *ref-forces the context* $\Gamma; \Delta$ *(and we write* $k \models \Gamma; \Delta$*) if*

1. *for every* $\varphi \in \Gamma$ *and any* $p \in P_k$*:* $(k, p) \models \Box\varphi$;
2. *for every* ψ **at** $q \in \Delta$*:* $q \in P_k$ *and* $(k, q) \models \psi$.

Finally, we extend the definition of forcing to judgements.

Definition 11 (Satisfaction for a Judgment). *Let* $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ *be a refined distributed Kripke model. We say that the judgement* $\Gamma; \Delta \vdash^P \mu$ ***at*** p is valid in $\mathcal{K}_{ref}$, *if*

- $PL(\Gamma) \cup PL(\Delta) \cup \{p\} \subseteq P$;
- *for every* $k \in K$ *such that* $P \subseteq P_k$, *if* $k \models \Gamma; \Delta$ *then* $k \models \mu$ **at** p.

Moreover we say that $\Gamma;\Delta \vdash^P \mu$ ***at*** p is ref-valid *(and we write* $\Gamma;\Delta \models \mu$ **at** p*) if it is valid in every refined distributed Kripke model.*

5.1 Soundness

In this section we shall prove the soundness of the extended logic in refined distributed Kripke models. The proof of soundness will follow the proof of the soundness in section 3.2. We start by defining the p-duplicated extension of a refined distributed Kripke model.

PROPOSITION 4 (p-DUPLICATED EXTENSION $\mathcal{K}_{ref}\langle p,q\rangle$). *Consider a refined distributed model* $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ *with Pls as set of places. Choose two places* p, q *such that* $p \in Pls$ *and* $q \notin Pls$*. Let* $\mathcal{K}_{ref}\langle p,q\rangle \stackrel{\text{def}}{=} (K', \leq', \{P'_k\}_{k\in K'}, \{I'_k\}_{k\in K'})$ *where*

- K' *is* K*;*
- $\leq'$ *is* $\leq$*;*
- P'_k *is* $P_k \cup \{q\}$ *if* $p \in P_k$*, and* P_k *otherwise;*
- $I'_k : Atoms \to Pow(P'_k)$ *is defined as*

$$r \in I'_k(A) \text{ iff } \begin{cases} r \in I_k(A) \ (\text{for } r \in P_k); \\ p \in I_k(A) \ (\text{for } r = q). \end{cases}$$

Then $\mathcal{K}_{ref}\langle p,q\rangle$ *is a refined distributed Kripke model, and is said to be a* p*-duplicated extension of* $\mathcal{K}$*.*

Proof: We just need to check that $\{P'_k\}_{k\in K'}$ and $\{I'_k\}_{k\in K'}$ satisfy the monotonicity conditions of Def. 8. They follow immediately from the definition of P'_k and I'_k. ■

We now show that the refined p-duplicated extension is conservative over all the formulae that do not mention the added place. Moreover, for all such formulae, the new place mimics the duplicated one.

LEMMA 7 ($\mathcal{K}_{ref}\langle p,q\rangle$ IS CONSERVATIVE). *Let* $\mathcal{K}_{ref}$ *be a refined distributed Kripke model with set of places, Pls, and* $\mathcal{K}_{ref}\langle p,q\rangle$ *be its* p*-duplicated extension. Let* $\models$ *and* $\models'$ *extend the interpretation of atoms in* $\mathcal{K}$ *and* $\mathcal{K}_{ref}\langle p,q\rangle$ *respectively. For every* $k \in K$ *and formula* $\varphi \in Frm(Pls)$*, we have:*

1. *for every* $r \in P_k$*,* $(k,r) \models' \varphi$ *if and only if* $(k,r) \models \varphi$*; and*
2. *if* $q \in P'_k$*, then* $(k,q) \models' \varphi$ *if and only if* $(k,p) \models \varphi$*.*

Proof: The proof is similar to the proof of Lemma 2 and we prove both properties simultaneously by induction on the structure of formulae in $Frm(Pls)$.

Base of induction. The two properties are easily verified on atoms and on $\top$ by the definition of p-duplicated extension.

Inductive hypothesis. We consider a formula $\varphi \in Frm(Pls)$ and assume that the two properties hold for every sub-formula of φ. In particular, we assume that if φ_i is a sub-formula of φ then *for every* $k \in K$*:*

1. *if* $r \in P_k$*, then* $(k, r) \models' \varphi_i$ *if and only if* $(k, r) \models \varphi_i$*; and*

2. *if* $q \in P'_k$*, then* $(k, q) \models' \varphi_i$ *if and only if* $(k, p) \models \varphi_i$*.*

The inductive cases for the connectives and modality @ have the same treatment as in Lemma 2. Here we show the most interesting cases, $\Box$ and $\Diamond$, by considering only property *1*. The treatment of property *2* is analogous. Pick $k \in K$ and $r \in P_k$, and fix them.

Case $\varphi = \Diamond\varphi_1$. Suppose $(k, r) \models' \Diamond\varphi_1$, then there is some $s \in P'_k$ such that $(k, s) \models \varphi_1$. In the case $s \in P_k$ we use induction to obtain $(k, s) \models \varphi_1$ and therefore $(k, r) \models \Diamond\varphi_1$. In the case $s = q$ we use induction to obtain $(k, p) \models \varphi_1$ and therefore $(k, r) \models \Diamond\varphi_1$. Vice versa, if $(k, r) \models \Diamond\varphi_1$ then there exists $s \in P_k$ such that $(k, s) \models \varphi_1$. Hence $(k, s) \models' \varphi_1$ by induction and we conclude $(k, r) \models' \Diamond\varphi_1$.

Case $\varphi = \Box\varphi_1$. Suppose that $(k, r) \models' \Box\varphi_1$. This means that $(l, s) \models' \varphi_1$ for every $l \geq k$ and every $s \in P'_l$. Since P'_l contains P_l, we obtain $(l, s) \models' \varphi_1$ for every $l \geq k$ and every $s \in P_l$. Hence, by induction $(l, s) \models \varphi_1$ for every $l \geq k$ and every $s \in P_l$, and we conclude that $(k, r) \models \Box\varphi_1$.

Vice versa if $(k, r) \models \Box\varphi_1$ then $(l, s) \models \varphi_1$ for every $l \geq k$ and every $s \in P_l$. By inductive hypothesis, we get that for every $l \geq k$ and $s \in P_l$, $(l, s) \models' \varphi_1$. If $q \notin Pk'_l$ for all $l \geq k$, then $P_l = P'_l$. In this case we conclude that $(k, r) \models' \Box\varphi$. On the other hand, if $q \in P'_l$ for some $l \geq k$, then it means that $p \in P_l$ and hence $(l, p) \models \varphi_1$. By induction (see property 2 of the proposition) $(l, q) \models' \varphi_1$, and we conclude $(k, r) \models' \Box\varphi_1$. ■

We now show that by renaming a place in a Kripke model, we do not change the set of valid formulae as long as the formulae do not mention renamed place or the fresh name.

Proposition 5 (p-Renaming $\mathcal{K}_{ref}\langle q/p\rangle$). *Let* $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ *be a refined distributed Kripke model with set of places Pls. For a place* $q \notin \mathcal{K}_{ref}$*, define* $\mathcal{K}_{ref}\langle q/p\rangle = (K', \leq, \{P'_k\}_{k\in K'}, \{I'_k\}_{k\in K'})$ *where*

- K' *is* K*;*

- $\leq'$ *is* $\leq$*;*

- P'_k *is* $(P_k \setminus \{p\}) \cup \{q\}$ *if* $p \in P_k$*, and* P_k *otherwise;*

- $I'_k : Atoms \to Pow(P'_k)$ *is defined*[1] *as*

$$r \in I'_k(A) \text{ iff } \begin{cases} r \in I_k(A) \ (\text{if } r \in P_k); \\ p \in I_k(A) \ (\text{if } r = q). \end{cases}$$

$\mathcal{K}_{ref}\langle q/p\rangle$ is a refined distributed Kripke model, and is said to be a p-renaming *of $\mathcal{K}_{ref}$.*

Proof: As for Proposition 4, we just need to check that $\{P'_k\}_{k\in K'}$ and $\{I'_k\}_{k\in K'}$ satisfy the monotonicity conditions. They follow immediately by definition. ■

Lemma 8 ($\mathcal{K}_{ref}\langle q/p\rangle$ is conservative). *Let $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ be a refined distributed Kripke model on $\mathcal{K}_{ref}\langle q/p\rangle$ be its p-renaming. Let $\models$ and $\models'$ extend the interpretation of atoms in $\mathcal{K}_{ref}$ and $\mathcal{K}_{ref}\langle q/p\rangle$ respectively. For every $k \in K$, formula $\varphi \in Frm(Pls)$, and $r \in Pk_k$ we have:*

1. *if $r \neq p$, then $(k, r) \models \varphi$ if and only $(k, r) \models' \varphi$; and*
2. *if $r = p$, then $(k, p) \models \varphi$ if and only if $(k, q) \models' \varphi$.*

Proof: The proof is by induction on the structure of formulae in $Frm(Pls)$, and is similar to the proof for Lemma 7. ■

We are now ready to prove that the semantics is sound for the judgements of the logic. We need to show that if a judgement is provable in the extended natural deduction system, then it is also valid with respect to refined distributed Kripke models.

Theorem 3 (Soundness). *If $\Gamma; \Delta \vdash^P \mu$ **at** p is derivable in the logic, then it is ref-valid.*

Proof: The proof is by induction on the number n of inference rules used in the derivation of the judgement of $\Gamma; \Delta \vdash^P \mu$ ***at*** p. The proof is similar to the proof of Theorem 1.

Base of induction ($n = 1$). If the the derivation consists of either the axiom L, or the axiom G, or rule $\top I$ we use the same argument as in the proof of Theorem 1. The case $\bot E$ follows by definition of the forcing relation.

Inductive hypothesis ($n > 1$). We assume that the theorem holds for any judgment that is deducible by applying less than n instances of inference rules. We consider a judgment $\Gamma; \Delta \vdash^P \mu$ ***at*** p which is derivable in the logic by using exactly n instances of inference rules.

We fix a model $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ with set of places Pls such that $P \subseteq Pls$, and let $\models$ be the extension of I_k. Let $k \in K$ be an arbitrary state such that $k \models \Gamma; \Delta$. Fix k. We need to show $(k, p) \models \mu$. For this we consider the last inference rule used to obtain $\Gamma; \Delta \vdash^P \mu$ ***at*** p and proceed by cases. The

[1] Note that it cannot be the case that $r = p$, since $p \notin P'_k$.

treatment of logical connectives is standard. The modalities @ and $\Diamond$ are teated as in Theorem 1. If the last inference rule used is $\Box E$, then the result follows from a simple application of the definition. The most interesting case is when $\Box I$ is the last inference used, and we discuss this case below.

Case $\Box I$. It must be case that μ is of the form $\Box\varphi$. Moreover $\Gamma;\Delta \vdash^{P+q} \varphi$ ***at*** q for some $q \notin P$ by using $n-1$ instances of the rules, and $\mathsf{PL}(\Gamma;\Delta)\cup\mathsf{PL}(\varphi) \subseteq P$. By induction we know that $\Gamma;\Delta \vdash^{P+q} \varphi$ ***at*** q is ref-valid. Without loss of generality, we can assume that $q \notin Pls$ (otherwise, we can rename q in Pls, using Lemma 8).

We prove that $k \Vdash \Box\varphi$ ***at*** p. The semantics of $\Box$ says that we need to show that $l \Vdash \varphi$ ***at*** r, for all $l \geq k$ and $r \in P_l$. Fix one $l \geq k$ and one $r \in P_l$, and consider the refined r-duplicated extension $\mathcal{K}_{ref}\langle r, q\rangle$. $\mathcal{K}_{ref}\langle r, q\rangle$ is a refined distributed Kripke model with set of places, $Pls \cup \{q\}$. Let $\Vdash'$ be the forcing relation on $\mathcal{K}_{ref}\langle r, q\rangle$.

From the hypothesis $k \Vdash \Gamma;\Delta$ and by Kripke monotonicity (Lemma 6) we get $l \Vdash \Gamma;\Delta$. Therefore, since $\mathcal{K}_{ref}\langle r, q\rangle$ is a r-duplicated extension, we get $l \Vdash' \Gamma;\Delta$ by using Lemma 7. Now, since $P + q \subseteq Pls \cup \{q\}$ we can use inductive hypothesis on $\mathcal{K}_{ref}\langle r, q\rangle$ to obtain $k \Vdash' \varphi$ ***at*** q. Using Lemma 7 once again, we conclude that $l \Vdash \varphi$ ***at*** r. Since l and r are arbitrary, we conclude that $k \Vdash \Box\varphi$ ***at*** p. ■

5.2 Completeness

In this section, we will show that the refined semantics is complete for the natural deduction presented in Section 4. The proof will follow the standard proofs of completeness for intuitionistic modal logic [18]. In the proof, we construct a canonical model. If a judgement is not provable, then it will be invalidated in one of the Kripke states of the canonical model.

Please note that the notion of provability can be extended on possible non-finite sets Σ of sentences, as in Section 3.3. We say that $\Gamma;\Sigma \vdash^P \varphi$ ***at*** p, if and only if, there exists a finite subset $\Delta \subseteq \Sigma$ such that $\Gamma;\Delta \vdash^P \varphi$ ***at*** p. Also, note that Proposition 3 stated in Section 3.3 can be extended to the logic with disjunctive connectives. The canonical model is defined by considering a particular kind of set of sentences.

Definition 12 (Refined Prime Set). *Let P be a set of places and Γ be a set of pure formulae in $Frm(P)$. A (possibly non-finite) set Σ of sentences with $\mathsf{PL}(\Sigma) \subseteq P$, is said to be* (Γ, P)-refined prime *if it satisfies the following four properties.*

1. *If* $\Gamma;\Sigma \vdash^P \varphi$ **at** p *then* φ **at** $p \in \Sigma$ (Deductive Closure).
2. $\Gamma;\Sigma \nvdash^P \bot$ **at** p *for any* $p \in P$ (Consistency).
3. *If* $\Gamma;\Sigma \vdash^P \varphi \vee \psi$ **at** p *then either* φ **at** $p \in \Sigma$ *or* ψ **at** $p \in \Sigma$ (Disjunction Property).
4. *If* $\Gamma;\Sigma \vdash^P \Diamond\varphi$ **at** p *then there exists* $q \in P$ *such that* φ **at** $q \in \Sigma$ (Diamond

Property).

As in [18, 2] we first show that every set of sentences can be extended to a prime set, that respects the non-provability with respect to a particular sentence.

LEMMA 9 (REFINED PRIME EXTENSION). *Let P be a set of places and Γ be a finite set of pure formulae in $Frm(P)$. Let φ be a pure formula, p be a place, and Δ be a set of sentences such that*

- $\mathsf{PL}(\varphi \textbf{ at } p) \cup \mathsf{PL}(\Delta) \subseteq P$, *and*
- $\Gamma; \Delta \nvdash^P \varphi \textbf{ at } p$.

Then there is a set of places P' extending P and a (Γ, P')- refined prime set of sentences Σ containing Δ, such that $\Gamma; \Sigma \nvdash^{P'} \varphi \textbf{ at } p$.

Proof: We enrich the set of places by introducing a denumerable set of new places: $\mathbf{q}_1, \mathbf{q}_2, \ldots$. They will be the witnesses for the formulae $\Diamond\varphi$ and are introduced in order to satisfy the diamond property.

The set of places P' is obtained by a series of extensions $P = P_0 \subseteq P_1 \subseteq P_2 \ldots$. Before we proceed with the construction, we pick up an enumeration of the pure formulae $Frm(P')$ and fix it. The set Σ is obtained by series of extensions $\Delta = \Sigma_0 \subseteq \Sigma_1 \subseteq \Sigma_2 \ldots$ that verify the following:

Property 2. For every $n \geq 0$:

1. $\mathsf{PL}(\Sigma_n) \subseteq P_n$;
2. $\Gamma; \Sigma_n \nvdash^{P_n} \varphi \textbf{ at } p$.

The series is constructed inductively. In the induction, at an odd step we will create a witness for a formula of the type $\Diamond\psi$. At an even step we deal with disjunction property. We shall also construct two sets:

- $treated_n^{\Diamond}$, that will be the set of the formulae $\Diamond\varphi$ for which we have already created a witness.
- $treated_n^{\vee}$, that will be the set of the formulae $\varphi \vee \psi$ ***at*** p which satisfy the disjunction property.

We start $treated_0^{\Diamond} = \emptyset$, $treated_0^{\vee} = \emptyset$, $P_0 = P$ and $\Sigma_0 = \Delta$. It is clear that $\mathsf{PL}(\Sigma_0) \subseteq P_0$, and $\Gamma; \Sigma_0 \nvdash^{P_0} \varphi$ ***at*** p.

Then we proceed inductively, and assume that P_n, Σ_n ($n \geq 0$) have been constructed satisfying Property 2. In step $n + 1$, we consider two cases:

1. If $n + 1$ is odd, pick the first formula $\psi_1 \vee \psi_2$ in the enumeration such that
   - $\psi_1 \vee \psi_2$ is in $Frm(P_n)$, i.e., all the places in $\psi_1 \vee \psi_2$ are taken from P_n;
   - $\Gamma; \Sigma_n \vdash^{P_n} \psi_1 \vee \psi_2$ ***at*** q, for some $q \in P_n$;

- $\psi_1 \vee \psi_2$ ***at*** $q \notin \mathit{treated}^{\vee}_n$.

Please note that if both $\Gamma; \Sigma_n, \psi_1$ ***at*** $q \vdash^{P_n} \varphi$ ***at*** p and $\Gamma; \Sigma_n, \psi_2$ ***at*** $q \vdash^{P_n} \varphi$ ***at*** p, then we can deduce $\Gamma; \Sigma_n \vdash^{P_n} \varphi$ ***at*** p. However, we have that Σ_n, P_n satisfy Property 2. Hence, it must be the case that either $\Gamma; \Sigma_n, \psi_1$ ***at*** $q \nvdash^{P_n} \varphi$ ***at*** p, or $\Gamma; \Sigma_n, \psi_2$ ***at*** $q \nvdash^{P_n} \varphi$ ***at*** p.

We define $\Sigma_{n+1} = \Sigma_n \cup \{\psi_1$ ***at*** $q\}$ if $\Gamma; \Sigma_n, \psi_1$ ***at*** $q \nvdash^{P_n} \varphi$ ***at*** p, and $\Sigma_{n+1} = \Sigma_n \cup \{\psi_2$ ***at*** $q\}$ otherwise. We define $P_{n+1} = P_n$. We get by construction that P_{n+1}, Σ_{n+1} satisfy Property 2. Finally, we let $\mathit{treated}^{\vee}_{n+1} = \mathit{treated}^{\vee}_n \cup \{\psi_1 \vee \psi_2$ ***at*** $q\}$ and $\mathit{treated}^{\Diamond}_{n+1} = \mathit{treated}^{\Diamond}_n$.

2. If $n+1$ is even, pick the first formula $\Diamond\psi$ in the enumeration such that

   - $\Diamond\psi$ is in $\mathit{Frm}(P_n)$, i.e., all the places in $\Diamond\psi$ are taken from P_n;
   - $\Gamma; \Sigma_n \vdash^{P_n} \Diamond\psi$ ***at*** q, for some $q \in P_n$;
   - $\Diamond\psi \notin \mathit{treated}^{\Diamond}_n$.

   Let $P_{n+1} = P_n + q_{(n+1)/2}$, $\Sigma_{n+1} = \Sigma_n \cup \{\psi$ ***at*** $\mathbf{q}_{(n+1)/2}\}$, $\mathit{treated}_{n+1} = \mathit{treated}_n \cup \{\Diamond\psi\}$ and $\mathit{treated}^{\vee}_{n+1} = \mathit{treated}^{\vee}_n$. We claim that $\Gamma; \Sigma_{n+1} \nvdash^{P_{n+1}} \varphi$ ***at*** p.

   If $\Gamma; \Sigma_{n+1} \vdash^{P_{n+1}} \varphi$ ***at*** p, then $\Gamma; \Sigma, \psi$ ***at*** $\mathbf{q}_{(n+1)/2} \vdash^{P+\mathbf{q}_{(n+1)/2}} \varphi$ ***at*** p. Since $\Gamma; \Sigma_n \vdash^{P_n} \Diamond\psi$ ***at*** q, we get $\Gamma; \Sigma_n \vdash^{P_n} \varphi$ ***at*** p by the inference rule $\Diamond E$. This contradicts the hypothesis on P_n, Σ_n. Hence $\Gamma; \Sigma_{n+1} \nvdash^{P_{n+1}} \varphi$ ***at*** p.

Therefore, we get by construction that P_n, Σ_n satisfy Property 2. We define $P' = \bigcup_{n \geq 0} P_n$, and $\Sigma = \bigcup_{n \geq 0} \Sigma_n$. Clearly $P \subseteq P'$, and $\Delta \subseteq \Sigma$. Moreover, using Property 2, we can easily show that $\Gamma; \Sigma \nvdash^{P'} \varphi$ ***at*** p. Finally, we show that Σ is a (Γ, P')-refined prime set.

1. (Disjunction Property) If $\Gamma; \Sigma \vdash^{P'} \psi_1 \vee \psi_2$ ***at*** q, then let n be the least number such that $\Gamma; \Sigma_n \vdash^{P_n} \psi_1 \vee \psi_2$ ***at*** q. Clearly, $\psi_1 \vee \psi_2$ ***at*** $q \notin \mathit{treated}^{\vee}_n$, and $\Gamma; \Sigma_m \vdash^{P_m} \psi_1 \vee \psi_2$ ***at*** q for every $m \geq n$. Eventually $\psi_1 \vee \psi_2$ ***at*** q has to be treated at some stage $h \geq n$. Hence, either ψ_1 ***at*** $q \in \Sigma_{h+1}$ or ψ_2 ***at*** $q \in \Sigma_{h+1}$. Therefore, ψ_1 ***at*** $q \in \Sigma$ or ψ_2 ***at*** $q \in \Sigma$.
2. (Diamond Property) If $\Gamma; \Sigma \vdash^{P'} \Diamond\psi$ ***at*** q, then let n be the least number such that $\Gamma; \Sigma_n \vdash^{P_n} \Diamond\psi$ ***at*** q. As in the previous case, we assert that $\Diamond\psi$ ***at*** q is treated for some even number $h \geq n$. We get ψ ***at*** $\mathbf{q}_{h/2} \in \Sigma$ by construction.
3. (Deductive Closure) If $\Gamma; \Sigma \vdash^{P'} \psi$ ***at*** q, then $\Gamma; \Sigma \vdash^{P'} \psi \vee \psi$ ***at*** q. The first case then gives us that ψ ***at*** $q \in \Sigma$.
4. (Consistency) If $\Gamma; \Sigma \vdash^{P'} \bot$ ***at*** q, then $\Gamma; \Sigma \vdash^{P'} \varphi@p$ ***at*** q by the inference rule $\bot E$. Therefore, $\Gamma; \Sigma \vdash^{P'} \varphi$ ***at*** p by $@E$, which contradicts our construction. Hence, $\Sigma; \Gamma \nvdash^{P'} \bot$ ***at*** q.

We conclude that Σ is a (Γ, P')- refined prime extending Δ such that $\Gamma; \Sigma \nvdash^{P'} \varphi$ ***at*** p. ■

Now, we define the refined canonical model. In the refined canonical model, Kripke states are prime sets of sentences.

Definition 13 (Refined Canonical Model). *Given a finite set Γ of pure formulae, we define the* Γ-refined canonical model *to be the quadruple $\mathcal{M}_{\Gamma Ref} = (M, \leq, \{P_l\}_{l\in M}, \{I_l\}_{l\in M})$, where:*

- *M is set of all pairs (Σ, P) such that P is a set of places, and Σ is a (Γ, P)-refined prime set.*
- *$(\Sigma_1, P_1) \leq (\Sigma_2, P_2)$ if and only if $\Sigma_1 \subseteq \Sigma_2$ and $P_1 \subseteq P_2$.*
- *$P_{(\Sigma,P)} \stackrel{\text{def}}{=} P$.*
- *$I_{(\Sigma,P)} : Atoms \longrightarrow Pow(P_{(\Sigma,P)})$ is defined by: $p \in I_{(\Sigma,P)}(A)$ iff A **at** $p \in \Sigma$.*

We now show that in the canonical model a sentence is forced by a Kripke state (Γ, Σ) if and only if it is contained in Σ.

Lemma 10 (Refined Canonical Evaluation). *Let Γ be a finite set of pure formulae.*

1. *The Γ-refined canonical model $\mathcal{M}_{\Gamma Ref} = (M, \leq, \{P_l\}_{l\in M}, \{I_l\}_{l\in M})$ is a refined distributed Kripke model.*
2. *Let Pls be the set of places of $\mathcal{M}_{\Gamma Ref}$, and $\models$ be the forcing relation in $\mathcal{M}_{\Gamma Ref}$. For every $(\Sigma, P) \in \mathcal{M}_{\Gamma Ref}$, every formula $\varphi \in Frm(Pls)$, and every place $p \in Pls$, $(\Sigma, P) \models \varphi$ **at** p if and only if φ **at** $p \in \Sigma$.*

Proof: Clearly all the properties required for a refined distributed Kripke model are verified. All we have to prove is the part *2* of the proposition. The proof is standard, and we proceed by induction on the structure of the formula $\varphi \in Frm(Pls)$. Here, we just illustrate the inductive case in which φ is $\Box\varphi_1$. In the inductive hypothesis, we assume that part*2* is valid on all sub-formulae of φ.

Case $\Box\varphi_1$. Assume that $(\Sigma, P) \models \Box\varphi_1$ ***at*** p. By definition, this means that for every (Σ', P') greater than (Σ, P) and for every $r \in P'$, it is the case that $(\Sigma', P') \models \varphi$ ***at*** r (and therefore φ ***at*** $r \in \Sigma'$ by inductive hypothesis).

Chose a new place $q \notin P$. We claim that $\Gamma; \Sigma \vdash^{P+q} \varphi_1$ ***at*** q. Suppose $\Gamma; \Sigma \nvdash^{P+q} \varphi_1$ ***at*** q. Then by Lemma 4, there is a set of places Q extending $P + q$ and a (Γ, Q)-refined prime set Σ' extending Σ such that $\Gamma; \Sigma' \nvdash^{Q} \varphi_1$ ***at*** q. That means φ_1 ***at*** $p \notin \Sigma'$. Since (Σ', Q) is greater than (Σ, P), we obtain a contradiction. Therefore we conclude that $\Gamma; \Sigma \vdash^{P+q} \varphi_2$ ***at*** q.

Using the inference rule $\Box I$, we get $\Gamma; \Sigma \vdash^{P} \Box\varphi_1 p$. Since Σ is a (Γ, P)-prime set, we get that means $\Box\varphi_1$ ***at*** $p \in \Sigma$.

Vice-versa, let $\Box\varphi_1$ ***at*** $p \in \Sigma$. Pick (Σ', Q) greater than (Σ, P). We need to show $(\Sigma', Q) \models \Box\varphi_1$ ***at*** p. We have that $\Sigma \subseteq \Sigma'$, and therefore $\Box\varphi_1$ ***at*** $p \in \Sigma'$. We can apply $\Box E$ to prove that $\Gamma, \Sigma' \vdash^{Q} \varphi$ ***at*** q for every $q \in Q$. By definition of the

canonical model, Σ' is (Γ, Q)-prime set. Therefore, we obtain $\varphi_1 \textbf{\textit{ at }} q \in \Sigma'$ for every $q \in Q$. Hence by inductive hypothesis, $(\Sigma', Q) \models \varphi_1 \textbf{\textit{ at }} q$ for every $q \in Q$. Since $P \subseteq Q$, we get $(\Sigma', Q) \models \Box\varphi_1 \textbf{\textit{ at }} p$. ■

We are now ready to prove completeness.

THEOREM 4 (REFINED COMPLETENESS). $\Gamma; \Delta \models {}^{P}\varphi \textbf{ at } p \Longrightarrow \Gamma; \Delta \vdash^{P} \varphi \textbf{ at } p$

Proof: Assume that $\Gamma; \Delta \models {}^{P}\varphi \textbf{\textit{ at }} p \Longrightarrow \Gamma; \Delta \vdash^{P} \varphi \textbf{\textit{ at }} p$. We have:

1. $\mathsf{PL}(\Gamma) \cup \mathsf{PL}(\Delta) \cup \{p\} \subseteq P$.
2. If $\mathcal{K}_{ref} = (K, \leq, \{P_k\}_{k\in K}, \{I_k\}_{k\in K})$ is a refined distributed Kripke model, then for every $k \in K$ such that $P \subseteq P_k$, $k \models \varphi \textbf{\textit{ at }} p$ whenever $k \models \Gamma; \Delta$.

We need to show that $\Gamma; \Delta \vdash^{P} \varphi \textbf{\textit{ at }} p$.

Assume that $\Gamma; \Delta \nvdash^{P} \varphi \textbf{\textit{ at }} p$. Then by Lemma 9, there is a set of places $P' \supseteq P$, and a (Γ, P')-refined prime set of sentences Σ containing Δ such that $\Gamma; \Sigma \nvdash^{P'} \varphi \textbf{\textit{ at }} p$. We get $\varphi \textbf{\textit{ at }} p \notin \Sigma$.

Now consider the Γ-canonical model $\mathcal{M}_{\Gamma Ref}$, and let $\models$ be the forcing relation in $\mathcal{M}_{\Gamma Ref}$. Consider the Kripke state (Σ, P'). Δ is contained in Σ, and therefore $(\Sigma, P') \models \Gamma; \Delta$ by Lemma10. By our assumption, we get $(\Sigma, P') \models \varphi \textbf{\textit{ at }} p$. By Lemma 10, we get $\varphi \textbf{\textit{ at }} p \in \Sigma$. We have just reach a contradiction. Therefore, we can conclude that $\Gamma; \Delta \vdash^{P} \varphi \textbf{\textit{ at }} p$. ■

6 Related Work

The logic studied in Section 2 was introduced in [9, 10], where it was used as the foundation of a type system for a distributed λ-calculus in the *propositions-as-types* paradigm. Although the authors of [9, 10] do discuss how the logic could be useful in distribution of resources, they have no corresponding model. The proof terms corresponding to modalities have computational interpretation in terms of remote procedure calls ($@p$), commands to broadcast computations to all nodes ($\Box$), and commands to use portable code ($\Diamond$). In [9], the authors also introduce a sequent calculus for the logic and prove that it enjoys cut elimination.

From a logical point of view, this logic can be viewed as a hybrid modal logic [16, 1]. A hybrid logic internalises the model in the logic by using modalities built from pure names [16, 1]. In [9, 10], the modality $@p$ gives the logic a hybrid flavour. Work on hybrid logics has been usually carried out in a classical setting, see the hybrid logics web page (`http://hylo.loria.fr/`). More recently, a first intuitionistic version of hybrid logics were investigated in [2].

There are several intuitionistic modal logics in the literature, and [18] is a good source on them. The modalities in [18] have a temporal flavour, and the spatial interpretation was not recognised then. There are no places in the Kripke states, and there is an accessibility relation on states that expresses the next step of a computation.

The work in [2] introduces the first intuitionistic version of hybrid logics. It investigates how to add names in constructive logics resulting in hybrid versions. A modal logic is hybridized by adding a new kind of propositional symbols: *nominals*. The nominals are the names in the logic. The authors extend the modal system of [18] by introducing nominals. They give a natural deduction system and a Kripke semantics for this logic. They prove soundness and completeness for the semantics, and also give a normalisation result for the natural deduction.

The extension given in Section 4 is a hybrid version of the intuitionistic modal system *IS5* [18]. In the modal system *IS5*, the accessibility relation among places is total. Hence, the logic in Section 4 can be seen as an instance of the hybrid modal logic in [2]. The only difference is that names in our logic only occur in the modality $@p$. In [2], names also occur as propositions.

Other work on logics in resources can be related to the separation logics [17], or the logic of bunched implications [15]. In [15], the authors give a Kripke model founded on a monoidal structure. In the logic, the formulae are the resources, and are interpreted as elements of the monoid. The focus of this work is the sharing of resources and not their distribution. There is no notion of places, and the logic has no modalities.

In the classical setting, there are also a number of logics used to study spatial properties. In [4, 3], for example, the authors use process calculi as their models. They have a classical modal logic to study spatial, temporal and security properties of the processes.

7 Conclusions and Future Work

We study the hybrid modal logic presented in [9, 10]. Formulae in the logic contain names, also called places. The logic may be used to reason about placement of resources in a distributed system. An intuitionistic natural deduction for this logic is presented in [10], and judgements mention the places under consideration.

We interpret the judgements in the logic in Kripke-style models [12]. Typically Kripke models [12] consists of partially ordered Kripke states. In our case the models are obtained from the Kripke models by adding a fixed set of places to each possible Kripke state. In each Kripke state, different places may satisfy different formulae. The satisfaction of atoms corresponds to placement of resources. The modalities of the logic allow formulae to be satisfied in a named place ($@p$), some place ($\Diamond$) and every place ($\Box$). We show that the interpretation of judgments in these models is both sound and complete.

We add disjunctive connectives to the modal logic in [9, 10], and refine our semantics to obtain soundness and completeness results. In the new Kripke models, larger Kripke states may contain bigger set of places. The refined semantics can be seen as an instance of hybrid *IS5* [2, 18].

As future work, we are currently investigating decidability of the extended logic. The intuitionistic modal systems in [18] are decidable. In order to prove decidability of those systems, [18] uses *birelational models*. These models are sound and complete, and enjoy finite model property: if a judgement is not valid in the logic, then there is a finite birelational model which invalidates the judgement. The finite model property is not enjoyed by the Kripke models in [18]. We are investigating if we can adapts the proofs in [18].

We are also considering other extensions of the logic. A major limitation of the logic presented in [10] is that if a formula φ is validated at some named place, say p, then the formula $\varphi@p$ can be inferred at every other place. Similarly if $\Diamond\varphi$ or $\Box\varphi$ can be inferred at one place, then they can be inferred at any other place. In a large distributed system, we may want to restrict the rights of accessing information in a place. This can be done by adding an accessibility relation as in [18, 2]. We are currently investigating the computational interpretation of this extended logic. This would result in an extension of λ-calculus presented in [9, 10].

ACKNOWLEDGEMENTS. We thank Annalisa Bossi, Giovanni Conforti, Matthew Hennessy, and Bernhard Reus for interesting and useful discussions.

References

[1] P. Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of the IGPL*, 8:339–365, 2000.

[2] T. Braüner and V. de Paiva. Towards constructive hybrid logic (extended abstract). In *Elec. Proc. of Methods for Modalities 3*, 2003.

[3] L. Caires and L. Cardelli. A spatial logic for concurrency (part I). In *TACS'01*, volume 2215 of *LNCS*, pages 1–37. Springer Verlag, 2001.

[4] L. Cardelli and A.D. Gordon. Anytime, anywhere. Modal logics for mobile ambients. In *POPL'00*, pages 365–377. ACM Press, 2000.

[5] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science, Special Issue on Coordination*, 240(1):177–213, 2000.

[6] J.-Y. Girard. *Proofs and Types*. Cambridge University Press, 1989.

[7] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[8] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

[9] L. Jia and D. Walker. Modal proofs as distributed programs. Technical Report TR-671-03, Princeton University, 2003.

[10] L. Jia and D. Walker. Modal proofs as distributed programs (extended abstract). In *ESOP'04*, volume 2986 of *LNCS*, pages 219–233. Springer Verlag, 2004.

[11] S.A. Kripke. Semantical analysis of modal logic I: Normal modal propositional calculi. In *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, volume 9, pages 67–96, 1963.

[12] S.A. Kripke. Semantical analysis of intuitionistic logic, I. In *Proc. of Logic Colloquium,*

Oxford, 1963, pages 92–130. North-Holland Publishing Company, 1965.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[14] J. Moody. Modal logic as a basis for distributed computation. Technical Report CMU-CS-03-194, Carnegie Mellon University, 2003.

[15] P.W. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[16] A. Prior. *Past, Present and Future*. Oxford University Press, 1967.

[17] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE Computer Society Press, 2002.

[18] A.K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[19] K. Crary T. Murphy, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS'04*, 2004. To appear.

[20] D. van Dalen. *Logic and Structure*. Springer Verlag, 4th extended edition, 2004.

BiLogics: Spatial-Nominal Logics for Bigraphs*

Giovanni Conforti[1] Damiano Macedonio[2] Vladimiro Sassone[3]

[1]Università di Pisa [2]Università Ca' Foscari di Venezia [3]University of Sussex

Abstract. Bigraphs are emerging as an interesting model for concurrent calculi, like Petri nets, pi-calculus, and ambients. Bigraphs are built orthogonally on two structures: a hierarchical place graph for locations and a link (hyper-)graph for connections. We introduce a logic for bigraphical structures as a natural composition of a place graph logic and a link graph logic. The former is a generalization of Spatial Tree Logic and the latter is a sort of Nominal Logic. We inspect the concepts of separation and sharing in these logics and some of their fragments.

1 Introduction

To describe and reason about structured, distribuited, and dynamic resources is one of the main goal of global computing research. Recently, many *Spatial Logics*, in different contexts, have been studied to fullfill this goal. The term 'spatial', as opposed to 'temporal', refers to the use of modal operators inspecting the spatial structure of the model. Spatial logics are usually equipped with a separation/composition binary operator that *splits* the current model into two parts, in order to 'talk' about them separately. Looking closely, we observe that notion of *separation* is interpreted differently in different logics. In 'separation' logics [21, 19], the separation is used to reason about dynamic update of heap-like structures, and it is *strong* in that it forces names of resources and pointers in separated components to be disjoint. As a consequence of this constraint, model composition is usually partially defined. In static spatial logics (e.g. for trees [3], graphs [5] or trees with hidden names [6]), the separation/composition operator is *structural*, and it is used to describe properties of the underlying structure. In this case no constraint on the model is usually required, and *names* may be shared between separated parts. In dynamic spatial logics (e.g. for ambients [7] or π-calculus [2]), the separation is intended only for location in space, and names can be shared between separated resources. Context tree logic, recently introduced in [4], integrates the first approach above with spatial logics for trees. The resulting logic is able to express properties of tree-shaped structures (and contexts) with pointers, and it is used as an assertion language for Hoare-style program specifications in a tree memory model.

Bigraphs [14, 16] are an emerging model for structures in global computing, which can be instantiated to model several well-known examples, including the π-calculus [13, 14], the ambient calculus [15] and Petri nets [18]. Bigraphs consist essentially of two graphs sharing the same nodes. The first graph, the *place graph*, is tree structured and expresses a hierarchical relationship on nodes (viz. locality in space and nesting of locations). The second graph, the *link graph*, is an hyper-graph and expresses a generic

* Research partially supported by '**MyThS**: Models and Types for Security in Mobile Distributed Systems', EU FET-GC IST-2001-32617 and by '**DisCo**: Semantic Foundations of Distributed Computation', EU IHP 'Marie Curie' HPMT-CT-2001-00290.

n-to-n relationships among nodes (e.g. data link, sharing of a channel). The two structures are orthogonal, so links between nodes can cross locality boundaries. Thus, bigraphs make clear the difference between structural separation (i.e. separation in the place graph) and name separation (i.e. separation on the link graph). By combining these two notion we obtain the 'strong' version of separation for general bigraphs.

In this paper we introduce a Spatial Logic for bigraphs as a natural composition of a place graph logic (for tree contexts) and a link graph logic (for name linkings). The main point is that a resource has a spatial structure as well as a link structure associated to it. Suppose for instance to be describing a tree-shaped distribution of resources in locations. We may use atomic formulae like $\mathsf{PC}(A)$ and $\mathsf{PC}_x(A)$ to describe a resource in an unnamed location, respectively location x, of 'type' PC (e.g. a computer) whose contents satisfy A. We can then write $\mathsf{PC}(\mathbf{T}) \otimes \mathsf{PC}(\mathbf{T})$ to characterize models with two unnamed PC resources whose contents satisfy the tautologic formula (i.e., with anything inside). Using named locations, as e.g. in $\mathsf{PC}_a(\mathbf{T}) \otimes \mathsf{PC}_b(\mathbf{T})$, we are able to express name separation, i.e., that names a and b are different. Furthermore, using link expressions we can force name-sharing between resources with formulae like:

$$\overbrace{\mathsf{PC}_a(\mathsf{In}_c(1) \otimes \mathbf{T}) \otimes \mathsf{PC}_b(\mathsf{Out}_c(1) \otimes \mathbf{T})}^{c}$$

This describes two PC with different names, a and b, sharing a link on a distinct name c, which models, e.g., a communication channel. Name c is used as input for the first PC and as an output for the second PC.

A bigraphical structure is, in general, a context with several holes and open links that can be filled by composition. This means that the logic can describe contexts for resources at no addition cost. We can then express formulae like $\mathsf{PC}_a(\mathbf{T} \otimes \mathsf{HD}(id_1 \text{ and } A))$ that describes a modular PC, where id_1 represents a 'pluggable' hole. Contextual resources have many important applications. In particular, the contextuality of bigraphs is useful to specify reaction rules, but it can also be used as a general mechanism to describe contexts of bigraphical (bigraph-shaped) data structures (cf. [11]).

Bigraphs are establishing themselves a truly general (meta)model of global systems, and appear to encompass several existing calculi and models (cf. [14, 18, 15]). *BiLog*, our bigraph logic, aims at achieving the same generality as a description language: as bigraphs specialize to particular models, we expect BiLog to specialize to powerful logics on these. In this sense, the contribution of this paper is to propose BiLog as a unifying language for the description of global resources. We will explore this path in future work, fortified by the positive preliminary results obtained for semistructured data [11]. The paper is organized as follows: §2 and §3 provide an crash course on bigraphs and their term-algebra axiomatization; §4 introduces BiLog and its model theory. Our main technical result is the encoding in BiLog of the spatial logics of [3] and [4].

2 An informal introduction to Bigraphs

Bigraphs formalize distributed systems by focusing on two of their main characteristics: locality and interconnections. A bigraph consists of a set of *nodes*, which may be nested

in a hierarchical tree structure (the so-called place graph), and have ports that may be connected to each other (and to names) by *links* (the so-called link graph). Place graphs express locality, i.e., the physical arrangement of the nodes. Link graphs are hypergraphs and formalize connections among nodes. The orthogonality of the two structures dictates that nestings impose no constrain upon interconnections.

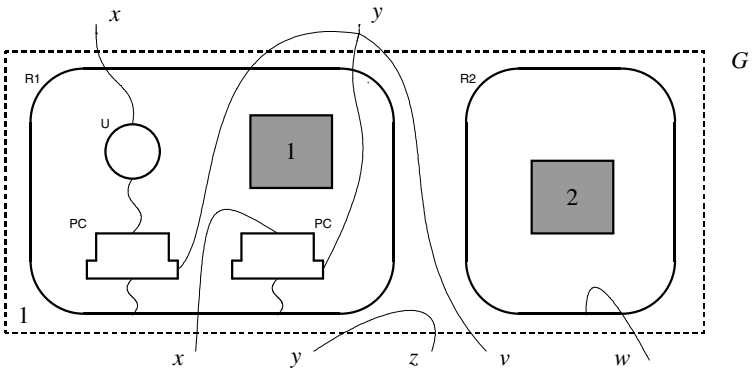


Fig. 1. A bigraph $G : \langle 2, \{x, y, z, v, w\}\rangle \to \langle 1, \{x, y\}\rangle$.

The bigraph G of Fig. 1 represents a system where people and things are interacting. We imagine two offices with employees logged on PCs. Every entity is represented by a node, shown with bold outlines, and every node is associated with a *control* (either PC, U, R1, R2). Controls represent kinds of nodes, and have fixed *arities* that determine their number of ports. Control PC marks nodes representing computers, and its arity is 3: in clockwise order, these ports represent a keyboard interacting with an employee U, a LAN connection interacting with other PCs and open to the global net, and a plug connecting the computer to the electrical system of the office R. Employees U may communicate with each other via the upper port in the picture. The nesting of nodes (place graph) is shown by the inclusion of nodes into each other, the connections (link graph) are drawn like lines.

At the top level of the nesting structure sit the *roots*. In Fig. 1 the root is unique (the dotted box). Inside nodes there may be 'context' *holes*, drawn as shaded boxes, which are uniquely identified by ordinals. In the figure the hole marked by 1 represents the possibility for another user U to get into the office R1 and sit in front of a PC. The hole marked by 2 represents the possibility to plug a subsystem inside office R2.

Place graphs can be seen as arrows over a symmetric monoidal category whose objects are ordinals. We write $P : m \to n$ to indicate a place graph P with m holes and n roots. The place graph of G in Fig.1 is of type $2 \to 1$. Given place graphs P_1, P_2, their composition $P_1 \circ P_2$ is defined only if the holes of P_1 are as many as the roots of P_2, and then amounts to *filling* holes with roots, orderly according to the number they carry. The tensor product $P_1 \otimes P_2$ is symmetric, but not commutative, as it 'renumbers' roots and holes 'from left to right'.

Link graphs are arrows of a partial monoidal category whose objects are (finite) sets of names. We assume a denumerable set Λ of names. A link graph is an arrow $X \to Y$, with $X, Y \subseteq \Lambda$. The set X represents the *inner* names (drawn at the bottom

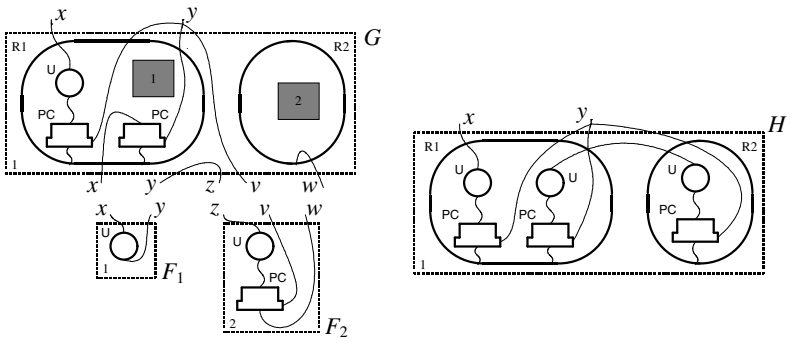


Fig. 2. Bigraphical composition

Fig. 2 represents a more complex situation. In its top left-hand side is the system of Fig. 1, in its bottom left-hand side are two other bigraphs: F_1 represents a user U ready to interact with a PC or with some other users, F_2 represents a user logged on its laptop and ready to communicate with other users. The system with F_1 and F_2 represents the tensor product $F = F_1 \otimes F_2$. The right-hand side of Fig. 2 represents the composition $G \circ F$. The idea is to insert F into the context G. The operation is partially defined, since it requires the inner names and the number of holes of G to match the number of roots and the outer names of F respectively. Shared names create the new links between the two structures. Intuitively, composition *first* places every root of F in the proper hole of G (place composition) and *then* joins equal inner names of G and outer names of F (link composition).

3 Bigraphical Terms

The models for our logic are the *(abstract pure) bigraphs* of [14]. As explained above, their interfaces are pairs $\langle m, X\rangle$, where m is an ordinal and X a set of names, and are equipped with a tensor product defined only when the corresponding sets of names are disjoint. In that case,

$$\langle m, X\rangle \otimes \langle n, X\rangle \stackrel{def}{=} \langle m + n, X \uplus Y\rangle\,.$$

In this paper we denote ordinals with m, n, s, names in Λ with a, b, sets of names with X, Y and interfaces with I, J. We define a bigraph to be *ground* if it has no holes (i.e., it has type $\langle 0, X\rangle \to \langle n, Y\rangle$), *prime* if it has no inner names and only one root (i.e., it is an arrow $\langle m, \emptyset\rangle \to \langle 1, Y\rangle$), and *discrete* if each of its links joins an outer name to exactly one port or inner name. Bigraphs F_1 and F_2 in Fig. 2 are prime discrete ground, and $F = F_1 \circ F_2$ is discrete ground.

The work [17] provides an algebraic axiomatization for bigraphs. Bigraphical terms are defined w.r.t. a control signature $\mathcal{K}$ and the set of names Λ. Signature $\mathcal{K}$ contains a set of controls K for nodes, each with a finite ordinal arity $ar(\mathsf{K})$. Abstract bigraphs G are formed from *elementary bigraphs* by composition and tensor product. We now introduce the elementary bigraphs, as they are the main inspiration for the logic.

Interfaces may degenerate in two forms: *place interfaces* $\langle m, \emptyset \rangle$ (that we write as m), and *link interfaces* $\langle 0, X \rangle$ (that we write as X, or just a if $X = \{a\}$). The *origin* $\epsilon = \langle 0, \emptyset \rangle$ is the unit for tensor product on interfaces. In Tab. 1 we list the elementary terms with their interfaces. Their explanation follows.

Table 3.1. *Bigraphical Terms*

Abstract Constant Placing Bigraphs				
G_p	::=	id_n	$: n \to n$	n holes with identity over names
		1	$: \epsilon \to 1$	empty single rooted bigraph
		$merge$	$: 2 \to 1$	merging in one root
		$\gamma_{m,n}$	$: n+m \to m+n$	swapping m with n places
Abstract Constant Linking Bigraphs				
G_l	::=	$/a$	$a \to \epsilon$	name closure (adding an edge)
		${}^{a}/_{X}$	$: X \to a$	substitution
Abstract Bigraphs				
G, F	::=	$\mathsf{K}_{\vec{a}}$	$: 1 \to \langle 1, \vec{a} \rangle$	discrete ion with $ar(\mathsf{K}) = \lvert\vec{a}\rvert$
		G_p		constant place bigraph
		G_l		constant link bigraph
		$G \otimes F$		tensor product
		$G \circ F$		composition

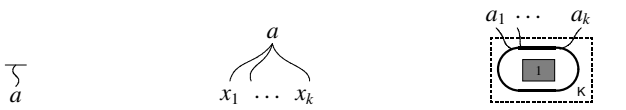
A *placing* is a bigraph $m \to n$ with no nodes. The place identity id_n is neutral for composition. All placings can be expressed in terms of the constants G_p reported in the table: 1 represents a barren root, *merge* maps two roots into one, $\gamma_{m,n}$ is a permutation that interchanges the first m places with the following n. Placings generated by composition and tensor product from $\gamma_{m,n}$ are *permutations* and are denoted by π.

A *linking* is a bigraph $X \to Y$ with no roots (and no holes). All linkings can be expressed in term of the constants G_l of Tab. 1. The closure $/a$ associates the name a with an edge so that the names is no more visible in the outerface. The substitution ${}^{a}/_{X}$ associates all the names in the set X to the name a. We denote linkings by ω, substitutions by σ, τ, and *renamings* (i.e., bijective substitutions) by α, β.

The final elementary bigraph is the *discrete ion* $\mathsf{K}_{\vec{a}}$ with $\vec{a} = a_1, \ldots, a_k$ and $k = ar(\mathsf{K})$, that is a single rooted bigraph consisting of a node with an hole inside; the node control is K, and each of its ports is connected to a different name in $\vec{a}$. In Fig. 3 we illustrate the two kind of linkings and the ion.

Clearly, not all combinations of elementary bigraphs are allowed, as composition must satisfy interfaces requirement and tensor product is allowed only in the case of no name-sharing. Term formation must follow the typing rules outlined in Tab. 2 .

Table 3.2. *Typing rules*



closure $/a$ substitution ${}^{a}/_{X}$ discrete ion $\mathsf{K}_{\vec{a}}$

Fig. 3. Elementary linking bigraphs and a discrete ion.

$$\frac{G : \langle m_1, X_1\rangle \to \langle n_1, Y_1\rangle \quad F : \langle m_2, X_2\rangle \to \langle n_2, Y_2\rangle \quad X_1 \cap X_2 = Y_1 \cap Y_2 = \emptyset}{G \otimes F : \langle m_1 + m_2, X_1 \uplus X_2\rangle \to \langle n_1 + n_2, Y_1 \uplus Y_2\rangle}$$

$$\frac{G : \langle s, Z\rangle \to \langle n, Y\rangle \quad F : \langle m, X\rangle \to \langle s, Z\rangle}{G \circ F : \langle m, X\rangle \to \langle n, Y\rangle}$$

The sound and complete axiomatization of bigraphs in [17] consists of the axioms in Tab. 1 listed in the appendix, required to hold only when both sides are defined. There, we also report the 'normal form' theorem A.1.

By restricting the set of elementary bigraph it is possible to generate important classes of bigraphs, in particular place graphs, wirings and discrete bigraphs. A place graph can be seen as an ordered list of roots hosting unordered trees, and can be generated by eliminating names and link operators in general bigraphs. Every place graph is expressed by compositions, identities and tensor product from the following constants: 1, id_n, *merge*, $\gamma_{m,n}$, K. In particular, K is a single discrete ion with a control of arity 0.

Wirings are bigraphs with no nodes, roots nor holes. They are generated by removing the tree structure and the nodes from bigraphs. Roughly speaking, they represent function on names. The corresponding terms are generated by the elemental terms not acting on the tree structure: $id_\emptyset$, $/a$, ${}^{a}/_{X}$. It is possible to further specialize substitution into three types:

$$a \stackrel{def}{=} {}^{a}/_{\emptyset} \qquad a \leftarrow b \stackrel{def}{=} {}^{a}/_{\{b\}} \qquad a \leftleftarrows b \stackrel{def}{=} {}^{a}/_{\{a,b\}}$$

Constants of kind a represent the introduction of a name a, term $a \leftarrow b$ represents the renaming of b to a, and finally $a \leftleftarrows b$ takes two names a and b and links (or fuse) them in the name a. Note that the identity id_X (i.e., the identity for interface X) can be easily derived using renaming and tensor product.

The fact that the tensor product is only defined on bigraphs with disjoint inner/outer sets of names makes of it a *spatial*, *separation* operator, in the sense that it separates the model into two distinct parts that cannot share names.

We generalize the composition between wirings even if their interfaces do not respect the typing rules. If $W : X \to Y$ and $W' : X' \to Y'$ with $Y \subset X'$, then

$$[W']W \stackrel{def}{=} (W' \otimes id_{X' \setminus Y}) \circ W$$

Moreover if $\vec{a} = a_1, \ldots, a_n$ and $\vec{b} = b_1, \ldots, b_n$, then $\vec{a} \leftarrow \vec{b}$ is a shortcut for $a_1 \leftarrow b_1 \otimes \ldots \otimes a_n \leftarrow b_n$ (and similarly for $\vec{a} \leftleftarrows \vec{b}$). It is possible to derive from the tensor product a new notion of parallel composition that allows name-sharing. Given $W : X \to Y$ and

$W' : X' \to Y'$ with $X \cap X' = \emptyset$, we consider $\vec{a} = Y \cap Y'$ (possibly not empty) and we choose a list $\vec{b}$ of fresh names not appearing in the models. The parallel composition is:

$$W \mid W' \stackrel{def}{=} [\vec{a} \leftleftarrows \vec{b}](([\vec{b} \leftarrow \vec{a}] \circ W) \otimes W')$$

The tensor product is well defined since all the common names $\vec{a}$ in W are renamed to fresh names, while the sharing is re-established afterwards by linking the $\vec{a}$ names with the $\vec{b}$ names. A discrete bigraph $D : \langle m, X\rangle \to \langle n, Y\rangle$ is a place graph with m holes and n roots together with a renaming producing an unique identifier $a_i \in Y$ for each port in the graph and inner name in X.

By specializing the general theorem of normal form for bigraphs, it is straightforward to obtain the characterizations for all the bigraphical classes introduced so far. The congruence is obtained by restricting the one of Tab. 1 on the classes. All the specific normal forms are outlined in the appendix.

4 BiLogics: Logics for Bigraphs

In this section, we define a propositional spatial logic for bigraphs by internalizing term constructors in the style of the ambient logic [7]. Elementary bigraphs are represented in the logic as constant formulae, while tensor product and composition are expressed by connectives. Therefore, we have two binary spatial operators. This contrasts with other spatial logics: the ambient logic, with only parallel composition $A \mid B$, the Separation Logic [21], with only the separating conjunction $A * B$, and the Context Tree Logic [4], with only the application $K(P)$. Both our operators inherit the monoidal structure and non-commutativity properties from the model.

Logical constants are classified in two classes: constants for locality, reflecting place graph terms, and constants for connections, reflecting linking terms. The sole formula expressing both resource locality and connections is the constant $\mathsf{K}_{\vec{a}}$, that reflects the corresponding discrete ion in the logic. In particular, if $\vec{a}$ is empty, K_ϵ represents a resource without identifier that cannot be linked to other resources, and does not allow any sharing. If all ions in a signature possess no ports (and hence no associated name), then the bigraphs on that signature actually are contextual trees, and can be described by the logical fragment without link constants (the *place graph logic*). On the other hand, a bigraph that has no root (and hence no node) can be completely described as a wiring by the logical fragment without placings (the *link graph logic*).

The semantics is given by a satisfaction relation between formulae and bigraphical terms, which is indexed by term types. We read the judgment $G \models_{I \to J} A$ as *"G is an arrow $I \to J$ and satisfies the formula A"*. Table 1 outlines the language and the semantics of the logic. We use $A, B, \ldots$ to range over formulae.

The satisfaction of logical constants is simply defined as the congruence to the corresponding elemental term and is self-explanatory. The formula $A \otimes B$ is satisfied by a bigraph G if G may be decomposed in the tensor product of two *separate* bigraphs such that A is satisfied by the former and B by the latter. Here 'separation' is intended to be *strong*, as in the separation logic. The connective $\otimes$ expresses both a separation on the structure and a separation on the associated names. In the following, we will see

Table 4.1. *BiLogic (located resources with names and sharing)*

$\eta, \eta' ::=$	a	name	x	name variable
$A, B ::=$	$\mathbf{F}$	false	$A \Rightarrow B$	implication
	$\mathbf{1}$	empty single rooted	$\mathbf{merge}$	merging two bigraphs
	$\mathbf{id}_{\langle m,\_\rangle}$	identity with m places	$\mathbf{id}_{\langle\_X\rangle}$	identity with names X
	$\gamma_{m,n}$	swapping the roots	$(\eta \leftarrow \eta')$	renaming
	$(\eta \Leftarrow \eta')$	link	$/\eta$	closure
	$\mathsf{K}_{\vec{\eta}}$	molecule K with ports $\vec{\eta}$	$\mathcal{N}x.A$	fresh name quantifier
	$A \otimes B$	tensor product	$A \circ B$	composition
	$A \circ\!\!- B$	left comp. adjunct	$A \multimap B$	right comp. adjunct

$$
\begin{array}{lcl}
G \models_{I\to J} \mathbf{F} & \stackrel{def}{=} & \text{never} \\
G \models_{I\to J} A \Rightarrow B & \stackrel{def}{=} & G : I \to J \text{ and } (G \models_{I\to J} A \Rightarrow G \models_{I\to J} B) \\
G \models_{I\to J} \mathbf{1} & \stackrel{def}{=} & G \equiv 1 \text{ and } I = \langle 0, \emptyset\rangle \text{ and } J = \langle 1, \emptyset\rangle \\
G \models_{I\to J} \mathbf{merge} & \stackrel{def}{=} & G \equiv \mathit{merge} \text{ and } I = \langle 2, \emptyset\rangle \text{ and } J = \langle 1, \emptyset\rangle \\
G \models_{I\to J} \mathbf{id}_{\langle m,\_\rangle} & \stackrel{def}{=} & \exists Y.\, G \equiv id_I \text{ and } I = J = \langle m, Y\rangle \\
G \models_{I\to J} \mathbf{id}_{\langle\_X\rangle} & \stackrel{def}{=} & \exists n.\, G \equiv id_I \text{ and } I = J = \langle n, X\rangle \\
G \models_{I\to J} \gamma_{m,n} & \stackrel{def}{=} & G \equiv \gamma_{m,n} \text{ and } I = \langle m+n, \emptyset\rangle \text{ and } J = \langle m'+n', \emptyset\rangle \\
G \models_{I\to J} (a \leftarrow b) & \stackrel{def}{=} & G \equiv a \leftarrow b \text{ and } I = \langle 0, b\rangle \text{ and } J = \langle 0, a\rangle \\
G \models_{I\to J} (a \Leftarrow b) & \stackrel{def}{=} & G \equiv a \Leftarrow b \text{ and } I = \langle 0, \{a,b\}\rangle \text{ and } J = \langle 0, a\rangle \\
G \models_{I\to J} /a & \stackrel{def}{=} & G \equiv /a \text{ and } I = \langle 0, a\rangle \text{ and } J = \langle 0, \emptyset\rangle \\
G \models_{I\to J} \mathsf{K}_{\vec{a}} & \stackrel{def}{=} & G \equiv \mathsf{K}_{\vec{a}} \text{ and } I = \langle 1, \emptyset\rangle \text{ and } J = \langle 1, \vec{a}\rangle \\
G \models_{I\to J} \mathcal{N}x.A & \stackrel{def}{=} & \exists a \notin I \cup J \cup fn(A).\, G \models_{I\to J} A\{x \leftarrow a\} \\
G \models_{I\to J} A \otimes B & \stackrel{def}{=} & \exists G_1 : I_1 \to J_1, G_2 : I_2 \to J_2.\, G \equiv G_1 \otimes G_2 \text{ and} \\
 & & G_1 \models_{I_1\to J_1} A \text{ and } G_2 \models_{I_2\to J_2} B \text{ and } I = I_1 \otimes I_2 \text{ and } J = J_1 \otimes J_2 \\
G \models_{I\to J} A \circ B & \stackrel{def}{=} & \exists G_1 : I' \to J, G_2 : I \to I'.\, G \equiv G_1 \circ G_2 \text{ and } G_1 \models_{I'\to J} A \\
 & & \text{and } G_2 \models_{I\to I'} B \\
G \models_{I\to J} A \circ\!\!- B & \stackrel{def}{=} & \forall G' : J \to J'.\, G' \models_{J\to J'} A \Rightarrow G' \circ G \models_{I\to J'} B \\
G \models_{I\to J} A \multimap B & \stackrel{def}{=} & \forall G' : I' \to I.\, G' \models_{I'\to I} A \Rightarrow G \circ G' \models_{I'\to J} B
\end{array}
$$

how this separation plays in various fragments of the logic. For instance, in the case of *Place Graph Logic*, where models are bigraphs without names, the separation is purely structural and coincides with the notion of parallel composition in Spatial Tree Logic. Dually, as the models for *Link Graph Logic* are bigraphs with no locations, the separation in such a logic is disjointness of names. Finally, for *Discrete Bigraph Logic*, where models' nodes are associated with names, but there are no links, the separation is not only structural, but also nominal, since the constraints on composition force port identifiers to be disjoint. In this sense, it can be seen as the separation in memory structures with pointers (like the heap structure of Separation Logic or the tree structure of Context Tree Logic).

Names can be internalized and effectively made private to a bigraph by the closure operator $/a$. The effect of composition with $/a$ is to add a new edge with no public name, and therefore to make a disappear from the outerface, and hence be completely hidden to the outside. Notice that separation is still expressed by the tensor connective, which not only separates places with an ideal line, but also makes sure that no edge – whether by visible or hidden – crosses the line. A fresh name quantifier inspired by Nominal Logic [20], will allows us to define the notion of separation with links.

A composition $A \circ B$ is satisfied by bigraphs that can be seen as a composed of two bigraphs that respectively satisfy A and B. We will see that the composition operator corresponds in some cases to well known spatial operators. We define the *left* and *right adjuncts* for composition to express extensional properties. The left adjunct $A \circ\!\!- B$ expresses that when the current bigraph is inserted in a context satisfying A, the resulting bigraph will then satisfy B. Dually, the right adjunct $A \multimap B$ expresses the property B that a (contextual) bigraph must fulfill whenever filled with any structure satisfying A.

We now show some basic results about BiLog and some of its fragments. In particular, we observe that the induced logical equivalences coibcide with the structural congruences for the terms they describe. This property is fundamental in order to describe, query and reason about bigraphical data structures, as e.g. XML (cf. [11]). In other terms, BiLog is *intensional* in the sense of [12] (i.e., it can observe internal structures), as opposed to the extensional logics used to observe the behaviour of dynamic system. Following [12], it would be possible to study a fragment of BiLog without intensional operators ($\otimes$, $\circ$, and constants).

We denote with $\mathcal{A}_{BGL}$ the set of BiLog formulae listed in Tab. 1. We identify three fragments, $\mathcal{A}_{PGL}, \mathcal{A}_{LGL}, \mathcal{A}_{DGL}$, that are suitable to describes place graphs, wirings, and discrete bigraphs respectively. Here we use $\mathcal{A}\{\ldots\}$ to denote is the language generated by constants and connectives in brackets.

$$\mathcal{A}_{PGL} \stackrel{def}{=} \mathcal{A}\{\mathbf{F},\ \Rightarrow,\ \mathbf{1},\ \mathbf{id}_{\langle \_,\emptyset\rangle},\ \mathbf{id}_{\langle n,\emptyset\rangle},\ \mathbf{merge},\ \gamma_{m,n},\ \mathsf{K},\ \otimes,\ \circ,\ \circ\!\!-,\ \multimap\}$$
$$\mathcal{A}_{LGL} \stackrel{def}{=} \mathcal{A}\{\mathbf{F},\ \Rightarrow,\ \mathbf{id}_{\langle 0,\_\rangle},\ a,\ (\eta \leftarrow \eta'),\ (\eta \leftleftarrows \eta'),\ /\eta,\ \otimes,\ \circ,\ \circ\!\!-,\ \multimap,\ \curlywedge\}$$
$$\mathcal{A}_{DGL} \stackrel{def}{=} \mathcal{A}\{\mathbf{F},\ \Rightarrow,\ \mathbf{1},\ \mathbf{id}_{\langle m,\_\rangle},\ \mathbf{id}_{\langle \_,X\rangle},\ \mathbf{merge},\ \gamma_{m,n},\ a,\ (a \leftarrow b),\ \mathsf{K}_{\vec{a}},\ \otimes,\ \circ,\ \circ\!\!-,\ \multimap\}$$

Note in particular that K in PGL has no names. The lemma below can be proved by induction on the structure of the formulae, and by applying the axioms in Tab.1.

Lemma 4.1 (Type and Congruence preservation).
If $G \models_{I\to J} A$ then $G : I \to J$.
If $G \equiv G'$ and $G \models_{I\to J} A$ then $G' \models_{I\to J} A$.

Every fragment of BiLog, and BiLog itself, induces a logical equivalence on the corresponding model in the obvious sense (the reader is referred to the Appendix for a formal treatment). It i easy to prove that the logical equivalence corresponds to the congruence in the corresponding model.

Theorem 4.2 (Logical equivalence is congruence).

$$\begin{array}{lll}
\textit{For every } G, G' : I \to J, & G =_{BGL} G' & \textit{if and only if } G \equiv G'.\\
\textit{For every } P, P' : m \to n, & P =_{PGL} P' & \textit{if and only if } P \equiv P'.\\
\textit{For every } W, W' : X \to Y, & W =_{LGL} W' & \textit{if and only if } W \equiv W'.\\
\textit{For every } D, D' : I \to J, & D =_{DGL} D' & \textit{if and only if } D \equiv D'.
\end{array}$$

Proof. The forward direction is proved by defining the characteristic formula of bigraphs (place graphs, link graphs, and discrete bigraphs) in normal form (cf. Appendix A). The converse is a direct consequence of Lemma 4.1. □

In the following section we examine the expressive power of BiLog by showing some interesting derived operators, and by providing encodings of other spatial logic into BGL fragments.

Table 4.2. *Derived Operators*

$\mathbf{T}$, and , $\vee, \Leftrightarrow \Leftarrow, \neg$ as usual for classical logics			
$\mathbf{id}$	$\stackrel{def}{=}$	$\mathbf{id}_{\langle 0,\_\rangle} \otimes \mathbf{id}_{\langle \_,\emptyset\rangle}$	Identity unconstrainted
$\mathbf{id}_{\langle m,X\rangle}$	$\stackrel{def}{=}$	$\mathbf{id}_{\langle m,\_\rangle} \circ \mathbf{id}_{\langle \_,X\rangle}$	Identity on interface $I = \langle m, X\rangle$
A_{I^*}	$\stackrel{def}{=}$	$A \circ \mathbf{id}_{I^*}$	Constrainting the inner face to be I^*
$A_{\rightarrow J^*}$	$\stackrel{def}{=}$	$\mathbf{id}_{J^*} \circ A$	Constrainting the outer face to be J^*
$A_{I^*\rightarrow J^*}$	$\stackrel{def}{=}$	$(A_{I^*})_{\rightarrow J^*}$	Constrainting the type to be $I^* \rightarrow J^*$
$A \circ_{I^*} B$	$\stackrel{def}{=}$	$A \circ \mathbf{id}_{I^*} \circ B$	Composition with interface I^*
$A \circ\!\!- _{J^*} B$	$\stackrel{def}{=}$	$A_{\rightarrow J^*} \circ\!\!- B$	Contexts with J^* as outer face
$A -\!\!\circ_{I^*} B$	$\stackrel{def}{=}$	$A_{I^*} -\!\!\circ B$	Composing with bigraphs with I^* as inner face
$A(B)$	$\stackrel{def}{=}$	$A \circ_1 B$	Simple application
$A \ominus B$	$\stackrel{def}{=}$	$\neg(\neg A \otimes \neg B)$	Dual of tensor product
$A \bullet B$	$\stackrel{def}{=}$	$\neg(\neg A \circ \neg B)$	Dual of composition
$A \bullet\!\!- B$	$\stackrel{def}{=}$	$\neg(\neg A \circ\!\!- \neg B)$	Dual of composition left adjunct
$A -\!\!\bullet B$	$\stackrel{def}{=}$	$\neg(\neg A -\!\!\circ \neg B)$	Dual of composition right adjunct

4.1 Derived Operators

In Table 2 we outline some interesting operators that can be derived in BiLogic. The operators constraining the interfaces are self-explanatory, where we write I^* and J^* for interfaces ranging over $\langle m, X\rangle$, $\langle m, \_\rangle$, and $\langle \_, X\rangle$. The 'dual' operators have the following semantics: $A \ominus B$ is satisfied by bigraphs G such that for every possible decompositon $G_1 \otimes G_2$ either $G_1 \models A$ or $G_2 \models B$. For instance, $A \ominus A$ describe bigraphs where A is true in (at least) one part of each $\otimes$-decomposition. The formula $\mathbf{F} \ominus (\mathbf{T}_{\rightarrow 1} \Rightarrow A) \ominus \mathbf{F}$ describes those bigraphs where every rooted component satisfies A. Similarly $A \bullet B$ expresses structural properties universally quantified on every decomposition of G as $G_1 \circ G_2$. They are both useful to specify security properties or types on bigraphical structures. The adjunct dual $A \bullet\!\!- B$ describes bigraphs that can be inserted into a context satisfying A – a sort of existential quantification on contexts – obtaining a bigraph satisfying B. For instance $(\mathbf{merge} \vee \mathbf{id}_1) \bullet\!\!- A$ describes the union between the class of double rooted bigraphs (with no names in the outerface) whose merging satisfies A, and the class of one rooted bigraphs satisfying A. Similarly the adjunct dual $A -\!\!\bullet B$ describes contextual bigraphs G such that there exists a bigraph satisfying A that can be inserted in G to obtain a bigraph satisfying B.

4.2 Place Graph Logic

In this section we focus on the place component of bigraphs. Place graphs can be interpreted as ordered lists of unordered trees with holes. We introduce the Place Graph Logic (PGL in the following) as the logic that maps place graph terms to spatial formulae. The logic, whose language is generated by $\mathcal{A}_{PGL}$, is essentially a multi-context logic on unordered trees. Not surprisingly, prime ground place graphs are isomorphic to the unordered trees that are models for the static fragment of ambient logic. Here we show that the logic restricted to ground prime place graphs can express the propositional spatial tree logic of [3].

By Theorem 4.2, it follows that PGL describes ground place graphs precisely, and the corresponding logical equivalence expresses structural congruence. Since place graphs

Table 4.3. *Encoding STL in PGL over prime ground place graphs*

Trees into Prime Ground Place Graphs

$[\![0]\!] \stackrel{def}{=} 1 \qquad [\![a[T]]\!] \stackrel{def}{=} \mathsf{K}(a) \circ [\![T]\!] \qquad [\![T_1 \mid T_2]\!] \stackrel{def}{=} merge \circ ([\![T_1]\!] \otimes [\![T_2]\!])$

STL formulae into PGL formulae

$[\![\mathbf{0}]\!] \stackrel{def}{=} 1 \qquad [\![a[A]]\!] \stackrel{def}{=} \mathsf{K}(a)([\![A]\!])$

$[\![\mathbf{F}]\!] \stackrel{def}{=} \mathbf{F} \qquad [\![A@a]\!] \stackrel{def}{=} \mathsf{K}(a) \circ\!\!- _1 [\![A]\!]$

$[\![A \Rightarrow B]\!] \stackrel{def}{=} [\![A]\!] \Rightarrow [\![B]\!] \qquad [\![A \mid B]\!] \stackrel{def}{=} [\![A]\!] \mid [\![B]\!]$

$[\![A \triangleright B]\!] \stackrel{def}{=} ([\![A]\!] \mid \mathbf{id}_1) \circ\!\!- _1 [\![B]\!]$

are constructed with a hierarchical structure, the logic resembles a propositional spatial tree logic, like [3]. The main difference are: (i) PGL models contexts of trees, so it has holes and the model-cheking is defined w.r.t. a number of roots and a number of holes; (ii) the tensor product is not strictly commutative, unlike the parallel composition, allowing us to model first level list-like structures.

PGL is able to describe tree contexts with many holes. Consider for instance a function that takes two trees and returns the tree obtained by merging their roots. This function is represented by the term *merge* and the formula **merge** is satisfied by this function solely. Similarly, the function that takes a tree and encapsulates it inside a node *labelled* by K, is represented by the term K and captured by the formula K. In general, the semantics of a formula A depends on the domain and codomain we are considering: we can constrain the domain in which it holds by using forms like $\mathbf{id}_n \circ A \circ \mathbf{id}_m$, which say that the current model must satisfy A and must be from m to n. As a further example, the formula $\mathbf{merge} \circ (\mathsf{K} \otimes (\mathbf{T} \circ \mathbf{id}_1))$ expresses all contexts of form $2 \to 1$ that put their first argument inside a K node and their second one as a sibling of such node.

The ordered tensor product is essential to model composition of contexts with several holes. For instance $(\mathsf{K}_1 \otimes \mathsf{K}_2) \circ (\mathsf{K}_3 \otimes 1)$ is different from $(\mathsf{K}_1 \otimes \mathsf{K}_2) \circ (1 \otimes \mathsf{K}_3)$, as node K_3 goes inside K_2 in the first case, and inside K_2 in the second one. However, we are able to define a commutative separation using **merge** and the tensor product: the *parallel composition* $A \mid B \stackrel{def}{=} \mathbf{merge} \circ (A_{\to 1} \otimes B_{\to 1})$. This separation is purely structural, and corresponds at term level to $merge \circ (P \otimes P')$ that is a total operation on all prime, one-rooted place graphs.

Propositional Spatial Tree Logic [3] (STL in the following) can be encoded in PGL using parallel composition. STL expresses properties of unordered labelled trees T constructed from the empty tree 0, the labelled node containing a tree $a[T]$, and the parallel composition of trees $T_1 \mid T_2$. It is a static fragment of the Ambient Logic [7] characterized by propositional connectives, spatial connectives (i.e., 0, $a[A]$, $A \mid B$), and their adjuncts (i.e., $A@a$, $A \triangleright B$). STL is quite expressive, even in the propositional case. For example, the adjunct operators express an implicit universal quantification on models, which can be used to internalize the validity problem in the model checking problem.

In Tab. 3 we encode models of STL (i.e., unordered trees) into prime ground place graphs, and STL operators into PGL operators. We assume a bijective encoding between labels and controls, and associate every label a with a distinct control $\mathsf{K}(a)$. The monoidal properties of parallel composition are guaranteed by the symmetry and unit axioms of *merge*. The logical encoding follows these lines, and the encoding of guarantee and location adjunct are obtained via the composition left adjunct.

The equations are self-explanatory once we remark that: *(i)* the parallel compostion of STL is the structural commutative separation of PGL; *(ii)* the tree labels can be represented by corresponding controls of the place graph; and *(iii)* that location and composition adjuncts of STL add logical expressible contexts to the tree, so they can be encoded in terms of the left composition adjunct.

Theorem 4.3 (Encoding STL). *For each tree T and formula A of STL we have that $T \models_{SL} A$ if and only if $[\![T]\!] \models_{0\to 1} [\![A]\!]$.*

Proof. The theorem is proved by using an *inverse encoding* $(\!|\ |\!)$ for prime ground place graphs to trees defined on normal forms. Composing $(\!|\ |\!)$ with $[\![\ ]\!]$ yields an isomorphism between trees and prime ground places graphs. The proof follows by structural induction on STL formulae. □

Besides expressing other known spatial logics, PGL can also describe structures with several holes and roots. In [11] we show some examples of how PGL can be used to describe (or even query) contexts of tree-shaped semistructured data. In particular multi-contexts can be useful to specify properties of web-service functions from (list of) XML trees to (list of) XML trees.

4.3 Link Graph Logic

In this section we focus on wiring bigraphs, which are a structured way to map a set of inner names into a set of outer names. Wirings can produce new names (idle names), α-convert existing names (renaming), connect two names and choose one as witness (link), or hide names (closure). We interpret wirings as composition of (partial) functions on sets of names, as an instance of more general algebraic structures (cf. [1]).

Link Graph Logic (PGL) is the language generated by $\mathcal{A}_{LGL}$, and its models are wirings. The formula $A \otimes B$ on wirings becomes a strong separation expressing that a wiring can be decomposed in two separate wirings (i.e., sharing no names nor connections) satisfying A and B respectively. Observe that in the special case of wirings, tensor is commutative as there are no ordered roots nor holes. In addition the separation has no structural component, since it can constrain the type of the model and not its structure. If we want a name a to be shared between separated resources, we need the sharing to be made explicit, and the sole way to do that is through the link operation. We therefore need a way to first separate the names occuring in two wirings in order to apply the tensor, and then link them back together. For this task, we employ a fresh name quantification in the style of Nominal Logic [20]. As a matter of fact, without name quantification is it not possible to build formulae that explore a link, since the latter has the effect of hiding names. Using fresh name quantification we can define a notion of a-linked name quantification for fresh names, whose purpose is to identify names that are linked to a:

$$a\mathbf{L}x.\, A \stackrel{def}{=} \mathcal{N}x.\, ((a \leftleftarrows x) \otimes \mathbf{id}) \circ A.$$

In the formula above, the role of $(a \leftleftarrows x)$ is to link the fresh name x whit a, while **id** deals with names different from a.

We also define a *separation-upto*, namely the decomposition in two models that are separated apart from the link on a specific name a, which crosses the separation line.

$$\overbrace{A \otimes B}^{a} \stackrel{def}{=} a\mathbf{L}x.(((x \leftarrow a) \otimes \mathbf{id}) \circ A) \otimes B.$$

The idea of the formula above is that the shared name a is renamed in a fresh name x, so that the product ca be performed and finally x is linked to a in order to actually have the sharing.

It would be possible to define a composition between wirings that is a total operation. This is actually what parallel composition for ambients and π-calculus processes do. However, such an operator would not by bifunctorial, as proved in [17]. This is due to the fact that the set of names shared by a parallel composition is not known in advance, and therefore parallel composition can only be defined using an existential quantification over the entire set of share names.

4.4 Discrete Bigraph Logic

PGL excels at expressing properties of *unnamed* resources, i.e., resources accessible only by following the structure of the model. On the other hand, LGL characterizes names and their links to resources, but it has no notion of locality or spatiality. If we want to model spatial structures with names, either private or public, we need to combine the two models. Bigraphs provide a good solution thanks to the orthogonal treatment of locality and connectivity. In this section, we make a first step in this direction by considering the Discrete Graph Logic (DGL). DGL is the fragment of BiLog generated by $\mathcal{A}_{DGL}$. Its models are the discrete bigraphs. The tensor product on discrete bigraphs is both a spatial separation (like in models of STL), and a partially defined separation on names (like that on pointers in separation logic). We define $*$ both as a term constructor and as a logical connective, as follows: $D * D' \stackrel{def}{=} [merge](D \otimes D')$, for D and D' discrete bigraphs, and $A * B \stackrel{def}{=} (\mathbf{merge} \otimes \mathbf{id}_{\langle 0,\_\rangle}) \circ (A_{\rightarrow\langle 1,\_\rangle} \otimes B_{\rightarrow\langle 1,\_\rangle})$, for A and B DGL formulae.

In [4] a spatial context logic is presented to reason about programs manipulating a tree structured memory with node identifiers used as pointers to memory locations. The complete structure has also link values, but here we restrict our attention to the fragment without them. Terms are unordered labelled trees T and unary contexts of trees C, i.e., trees with one hole. Correspondingly, the logic is defined by formulae of two kinds: formulae P, which describe trees, and formulae K, which describe tree contexts. Both of these have spatial operators built by using constants (i.e., the empty tree for T and the hole in C), application $K(P)$, and its two adjuncts $K \triangleright P$ and $P \triangleleft P$. Formula $K \triangleright P$ represents a tree that satisfies P if inserted in a context satisfying K. Dually, $P_1 \triangleleft P_2$ represent contexts that composed with a tree satisfying P_1 produce a tree satisfying P_2. It is easy to encode this structure in DGL by lifting the application to a particular kind of composition, and similarly for the two adjuncts. This encoding shows how the DGL is a generalization of Context Tree Logic to contexts with several holes (and roots). The encodings are detailed in Tab. 4.

Table 4.4. *Encoding Context TL in PGL over prime place graphs*

Trees into prime ground discrete bigraphs	Contexts into unary discrete bigraphs
$[\![0]\!] \stackrel{def}{=} 1$	$[\![-]\!]_C \stackrel{def}{=} id_1$
$[\![a_x[T]]\!] \stackrel{def}{=} (\mathsf{K}(a)_x \otimes fn(T)) \circ [\![T]\!]$	$[\![a_x[C]]\!]_C \stackrel{def}{=} (\mathsf{K}(a)_x \otimes fn(C)) \circ [\![C]\!]_C$
$[\![T_1 \mid T_2]\!] \stackrel{def}{=} [\![T_1]\!] * [\![T_2]\!]$	$[\![T \mid C]\!]_C \stackrel{def}{=} [\![T]\!] * [\![C]\!]_C$
	$[\![C \mid T]\!]_C \stackrel{def}{=} [\![C]\!]_C * [\![T]\!]$
TL formulae into PGL formulae	**Context formulae into PGL formulae**
$[\![false]\!]_P \stackrel{def}{=} \mathbf{F}$	$[\![false]\!]_K \stackrel{def}{=} \mathbf{F}$
$[\![K(P)]\!]_P \stackrel{def}{=} [\![K]\!]_K \circ_{\langle 1,\_\rangle} [\![P]\!]_P$	$[\![-]\!]_K \stackrel{def}{=} \mathbf{id}_1$
$[\![K \triangleleft P]\!]_P \stackrel{def}{=} [\![K]\!]_K \circ\!\!-_{\langle 1,\_\rangle} [\![P]\!]_P$	$[\![P \triangleright P']\!]_K \stackrel{def}{=} [\![P]\!]_P \multimap_{\langle 1,\_\rangle} [\![P']\!]_P$
$[\![P \Rightarrow P']\!]_P \stackrel{def}{=} [\![P]\!]_P \Rightarrow [\![P']\!]_P$	$[\![a_x[-]]\!]_K \stackrel{def}{=} \mathsf{K}(a)_x$
	$[\![P \mid -]\!]_K \stackrel{def}{=} [\![P]\!]_P * \mathbf{id}_1$
	$[\![K \Rightarrow K']\!]_K \stackrel{def}{=} [\![K]\!]_K \Rightarrow [\![K']\!]_K$

Theorem 4.4 (Encoding Context Tree Logic). *For each tree T and formula P of TL we have that $T \models_T P$ if and only if $[\![T]\!] \models_{0 \to \langle 1, fn(T)\rangle} [\![P]\!]_P$. Also, for each context C and formula K of Context TL we have that $C \models_K K$ if and only if $[\![C]\!]_C \models_{1 \to \langle 1, fn(C)\rangle} [\![K]\!]_K$.*

Proof. By structural induction on TL and Context TL formulae by defining an inverse encoding on normal forms for prime place graphs with one hole.

The encoding shows that the models introduced in [4] are a particular kind of discrete bigraphs with one port for each node and a number of holes and roots limited to one. Since [4] is more general than separation logic, and is used to reason about programs that manipulate tree structured memory model, we can express separation logic too and use DGL to reason about programs with complex query/update memory instruction, involving many locations contemporarily.

5 Conclusion and future work

In this paper we move a first step towards describing global resources by focusing on static bigraphs. We introduce BiLog, a logic for bigraphical structures, with two two main spatial connectives: composition and tensor product. Our main technical results are a comparison with other spatial logics previously studied, and the characterization of logical equivalence as term congruence. We identified two fragments of BiLog suitable to encode the propositional static fragment of Ambient Logic [3] and the latest Context Tree Logic [4].

We did not introduce the tensor adjuncts and existential/universal quantifiers. The formers are not necessary for the encodings, and it is an open issue whether or not they are definable BiLog. The quantifications are omitted as they imply an undecidable satisfaction relation (cf. [9]), while we aim at a decidable logic. As a matter of fact, we are working on extending the result of [3], and we are isolating decidable fragments of BiLog. We introduced the freshness quantifier as it is useful to express hiding and it preserves decidability in spatial logics [10]. In order to compare BiLog with other spatial logics thoroughly, we are developing sequent calculus.

In the paper we have not addressed a logic for tree with hidden names. As a matter of fact, we have such a logic. More precisely we can encode abstract trees into bigraphs with an unique control amb with arity one. The name assigned to this control will be actually the name of the ambient. The extrusion properties and renaming of abstract trees have their correspondence in bigraphical terms by means of substitution and closure properties combined with properties of identity.

BiLog can express properties of trees with names, like *DGL*, separation with links, like in *LGL*, and trees with links. At the logical level we may encode operators of tree logic with hidden names as follows:

$$\begin{aligned}
\copyright a &\stackrel{def}{=} ((a \leftarrow a) \otimes \mathbf{id}) \circ \mathbf{T} \\
\mathbf{C}x.\, A &\stackrel{def}{=} \mathcal{И}x.\, (/x \otimes \mathbf{id}) \circ A \\
a \circledR A &\stackrel{def}{=} (\neg \copyright a \wedge A) \vee (/a \otimes \mathbf{id}) \circ A \\
\mathbf{H}x.\, A &\stackrel{def}{=} \mathcal{И}x.\, x \circledR A
\end{aligned}$$

The operator $\copyright a$ says that the name a appears in the outer face of the bigraphs. The new quantifier $\mathbf{C}x.\, A$ expresses the fact that in a process satisfying A a name has been closed. The revelation $\circledR$ is a binary operator asserting the possibility of revealing a restricted name as a in order to assert A, note that the name may be hidden in the model as it has either be closed with an edge or it does not appear in the model. The hiding quantification $\mathbf{H}$ may be derived as in [8].We are currently working on the expressivity and decidability of this logical framework.

Several important questions remain: as bigraphs have an interesting dynamics, specified using reactions rules, we plan to extend BiLog to such a framework. Building on the encodings of the ambient and the π calculi into bigraphical reactive systems, we expect a dynamic BiLog to be able to express both ambient logic [7] and spatial logics for π-calculus [2].

Acknowledgment. We would like to thank Philippe Bidinger, Rohit Chadha, Murdoch Gabbay, Giorgio Ghelli, Robin Milner, and Peter O'Hearn for useful discussions.

References

1. R. Bruni, F. Gadducci, and U. Montanari. Normal forms for algebras of connections. *Theoretical Computer Science*, 286(2), September 2002.
2. L. Caires and L. Cardelli. A spatial logic for concurrency (Part I). In *Proc. of Theoretical Aspects of Computer Software; 4th International Symposium, TACS 2001*, volume 2215 of *LNCS*, pages 1–37. Springer-Verlag, 2001.
3. C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding validity in a spatial logic for trees. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, 2003.
4. C. Calcagno, P. Gardner, and U. Zarfaty. A context logic for tree update. In Proc. of LRPP 2004, revised version to appear in POPL 2005.
5. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *Proc. of ICALP*, volume 2380 of *LNCS*, page 597. Springer-Verlag, 2002.
6. L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *Proc. of FOSSACS '03*, volume 2620 of *LNCS*, pages 216–232. Springer-Verlag, 2003.

7. L. Cardelli and A. D. Gordon. Ambient logic. To appear in Mathematical Structures in Computer Science.
8. L. Cardelli and A. D. Gordon. Logical properties of name restriction. In *International Conference on Typed Lambda Calculi and Applications (TCLA 2001, Krakow, Poland)*, volume 2044 of *LNCS*, pages 46–60. Springer, 2001.
9. W. Charatonik and J.M. Talbot. The decidability of model checking mobile ambients. In *CSL: 15th Workshop on Computer Science Logic*, volume 2142 of *LNCS*, page 339, 2001.
10. G. Conforti and G. Ghelli. Decidability of freshness, undecidability of revelation. In *Foundations of Software Science and Computation Structures, FOSSACS 2004*, pages 105–120. Springer, 2004.
11. G. Conforti, D. Macedonio, and V. Sassone. Bigraphical logics for xml. Unpubilshed notes. `http://www.di.unipi.it/~confor/publications.html`, October 2004.
12. D. Hirschkoff. An extensional spatial logic for mobile processes. In *CONCUR - Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 2004.
13. O. H. Jensen and R. Milner. Bigraphs and transitions. In *Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–49. ACM Press, 2003.
14. O. H. Jensen and R. Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580. University of Cambridge, February 2004.
15. O.H. Jensen. Forthcoming PhD Thesis. Aalborg University, 2004.
16. R. Milner. Bigraphical reactive systems. In *Proc. of the 12th International Conference on Concurrency Theory*, volume 2154 of *LNCS*, pages 16–35. Springer, 2001.
17. R. Milner. Axioms for bigraphical structure. Technical Report UCAM-CL-TR-581. University of Cambridge, February 2004.
18. R. Milner. Bigraphs for petri-nets. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, pages 686–701. Springer, 2004.
19. Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *In Proc. of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer-Verlag, 2001.
20. A. M. Pitts. Nominal logic: A first order theory of names and binding. In *Proc. of TACS 2001*, volume 2215 of *LNCS*, pages 219–242. Springer-Verlag, 2001.
21. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74. IEEE Computer Society, 2002.

A Axioms, Normal Forms and characteristic formulae

The work in [17] presents a set of axioms for bigraphical terms. The categorical axioms are standard for a strict symmetric monoidal category. The composition is partially defined as usual, moreover the tensor product is defined only when interfaces have disjoint name sets. Thus the equations are required to hold only when both sides are defined. The expression considered in Tab. 1 are built by composition, identities and tensor product from the six class of elemental bigraphs described previously. Swappings $\gamma_{m,n}$ are extended to arbitrary interfaces in the following way:

$$\gamma_{I,J} \stackrel{def}{=} \gamma_{m,n} \otimes id_{X \uplus Y} \text{ where } I = \langle m, X \rangle\,,\ J = \langle n, Y \rangle$$

Table A.1. *Axioms*

Categorial Axioms:

$G \circ id \equiv G \equiv id \circ G$	Identity
$(G_1 \circ G_2) \circ G_3 \equiv G_1 \circ (G_2 \circ G_3)$	Associativity
$G \otimes id_\epsilon \equiv G \equiv id_\epsilon \otimes G$	Monoid Neutral Element
$(G_1 \otimes G_2) \otimes G_3 \equiv G_1 \otimes (G_2 \otimes G_3)$	Monoid Associativity
$(G_1 \otimes F_1) \circ (G_2 \otimes F_2) \equiv (G_1 \circ G_2) \otimes (F_1 \circ F_2)$	Bifunctoriality
$\gamma_{I,\epsilon} \equiv id_I$	Symmetry Id
$\gamma_{I,J} \circ \gamma_{J,I} \equiv id_{I \otimes J}$	Symmetry Composition
$\gamma_{I',J'} \circ (G \otimes F) \equiv (F \otimes G) \circ \gamma_{I,J}$	Symmetry Monoid
Link Axioms:	
$/a \circ {}^a/_b \equiv /b$	Closing renaming
$/a \circ a \equiv id_\epsilon$	Idle edge
${}^b/_{(Y \uplus a)} \circ (id_Y \otimes {}^a/_X) \equiv {}^b/_{Y \uplus X}$	Composing substitutions
Place Axioms:	
$merge \circ (1 \otimes id_1) \equiv id_1$	Unit
$merge \circ (merge \otimes id_1) \equiv merge \circ (id_1 \otimes merge)$	Associativity
$merge \circ \gamma_{1,1} \equiv merge$	Commutativity
Node Axiom:	
$(id_1 \otimes \alpha) \circ \mathsf{K}_{\vec{a}} \equiv \mathsf{K}_{\alpha(\vec{a})}$	Renaming

Moreover, the work in [17] provides the following theorem of normal form. In its statement, the bigraph $merge_n$ is defined as follows:

$$merge_0 \stackrel{def}{=} 1$$
$$merge_{n+1} \stackrel{def}{=} merge \circ (id_1 \otimes merge_n)$$

Theorem A.1 (Discrete Normal Form). *Every bigraph can be written in the following normal form solely up to renaming and permutations: $G = (id_n \otimes w)D$ where D is a discrete bigraph of the form $D = ((Q_0 \otimes \ldots \otimes Q_n)\pi) \otimes \alpha$ where Q_i are prime bigraphs of the form $Q = (merge_{n+k} \otimes id_Y)(id_n \otimes M_1 \ldots \otimes M_k)\pi$ where M_i are discrete molecule of the form $M = (\mathsf{K}_{\vec{a}} \otimes id_Y)Q$.*

A first consequence of Theorem A.1 is Tab. 2. Moreover the theorem can be instantiate in order to obtain normal forms for some bigraphical subclasses.

Corollary A.2 (Discrete Normal Form for Place Graph). *Every place graph P with outer width n may be expressed as $P = (Q_0 \otimes \ldots \otimes Q_{n-1}) \circ \pi$, where every Q_i is a prime place graph of the form $Q = merge_{n+s} \circ (id_n \otimes M_0 \otimes \ldots \otimes M_{s-1}) \circ \pi$, with M_j molecular place graphs of the form $M = \mathsf{K} \circ Q$ and $ar(\mathsf{K}) = 0$. This normal form is unique, modulo permutations.*

Corollary A.3 (Normal Form for Prime Ground Place Graph). *Every prime ground place graph $q : 0 \to 1$ may be expressed as $q = merge_s \circ (M_0 \otimes \ldots \otimes M_{s-1})$, with M_j molecular prime ground place graphs of the form $M = \mathsf{K} \circ q$ and $ar(\mathsf{K}) = 0$. This normal form is unique, modulo permutations.*

Table A.2. *Normal Forms and characteristic formulae*

Normal forms for bigraphs G, wirings W and discrete bigraphs D:

$G = (id_n \otimes W) \circ D$

$W = (/a_1 \otimes \ldots \otimes /a_k \otimes b_1 \otimes \ldots \otimes b_s \otimes \alpha) \circ \sigma$

$D = ((Q_1 \otimes \ldots \otimes Q_k) \circ \pi_k) \otimes \alpha$

$Q = [merge_{n+k}](id_n \otimes M_1 \otimes \ldots \otimes M_k)$

$M = [\mathsf{K}_{\vec{a}}]Q$

$\alpha = (a_1 \leftarrow b_1) \otimes \ldots \otimes (a_k \leftarrow b_k)$

$\sigma = {}^{a_1}\!/_{X_1} \otimes \ldots \otimes {}^{a_k}\!/_{X_k}$ with $a_i \in X_i$

$\pi_k = \gamma_{n,k-n} \circ (id_1 \otimes \pi_{k-1})$ with $0 \leq n < k$

Characteristic formulae for bigraphs G,wirings $W$and discrete bigraphs Dsf :

$\overline{G} = (\mathbf{id}_n \otimes \overline{W}) \circ \overline{D}$

$\overline{W} = (/a_1 \otimes \ldots \otimes /a_k \otimes b_1 \otimes \ldots \otimes b_s \otimes \overline{\alpha}) \circ \overline{\sigma}$

$\overline{D} = ((\overline{Q_1} \otimes \ldots \otimes \overline{Q_k}) \circ \overline{\pi_k}) \otimes \overline{\alpha}$

$\overline{Q} = (\mathbf{merge}_{n+k} \otimes \mathbf{id})(\circ\, \mathbf{id}_n \otimes \overline{M_1} \otimes \ldots \otimes \overline{M_k})$

$\overline{M} = (\mathsf{K}_{\vec{a}} \otimes \mathbf{id}) \circ \overline{Q}$

$\overline{\alpha} = ((a_1 \leftarrow b_1)) \otimes \ldots \otimes ((a_k \leftarrow b_k))$

$\overline{\sigma} = \overline{{}^{a_1}\!/_{X_1}} \otimes \ldots \otimes \overline{{}^{a_k}\!/_{X_k}}$ with $a_i \in X_i$

$\overline{\pi_k} = \gamma_{n,k-n}(\circ\, id_1 \otimes \overline{\pi_{k-1}})$ with $0 \leq n < k$

$\overline{{}^{a}\!/_{\{a\}}} = (a \leftarrow a)$

$\overline{{}^{a}\!/_{\{a,b\}}} = (a \leftleftarrows b)$

$\overline{{}^{a}\!/_{\{a,b\} \uplus X}} = (a \leftleftarrows b) \circ (\overline{{}^{b}\!/_{\{b\} \uplus X}} \otimes (a \leftarrow a))$

Normal form for Place Graphs P, a particular case with $\alpha = id_\epsilon$:

$P = ((Q_1 \otimes \ldots \otimes Q_k) \circ \pi_k)$

$Q = merge_{n+k} \circ (id_n \otimes M_1 \otimes \ldots \otimes M_k)$

$M = \mathsf{K} \circ Q$

$\pi_k = \gamma_{n,k-n} \circ (id_1 \otimes \pi_{k-1})$ with $0 \leq n < k$

Corollary A.4 (Normal Form for Discrete Bigraphs). *Every discrete bigraph can be written in the following normal form solely up to permutations:* $D = (Q_0 \otimes \ldots \otimes Q_n)\pi \otimes \alpha$ *where* Q_i *are prime bigraphs of the form* $Q = (merge_{n+k} \otimes id_Y)(id_n \otimes M_1 \otimes \ldots \otimes M_k)\pi$ *where* M_i *are discrete molecule of the form* $M = (\mathsf{K}_{\vec{a}} \otimes id_Y)Q$.

Corollary A.5 (Normal Form for Unary and Ground/Prime Discrete Bigraphs). *Every ground prime discrete bigraph* $q : 0 \to \langle 1, Y \rangle$ *can be written in the following normal form solely* $q = (merge_k \otimes id_Y) \circ (M_1 \otimes \ldots \otimes M_k)$ *where* M_i *are discrete ground molecule of the form* $M = (\mathsf{K}_{\vec{a}} \otimes id_Y)q$. *Every unary (with one hole and one root) discrete bigraph* $D : \langle 1, X \rangle \to \langle 1, Y \rangle$ *can be written in the following normal form solely:* $D = Q \otimes \alpha$ *where* $Q = (merge_k \otimes id_Y) \circ (R \otimes M_1 \otimes \ldots \otimes M_{k-1})$ *where* M_i *are discrete ground molecule and* R *can be either* id_1 *or* $(\mathsf{K}_{\vec{a}} \otimes id_Y) \circ Q$ *(a molecule with one hole inside). These normal form are unique up to permutations.*

The logical equivalence is defined as usual w.r.t. the language considered and the corresponding model.

Definition A.6 (Logical equivalence).

$$
\begin{array}{rcl}
G =_{BGL} G' & \stackrel{def}{=} & \forall A \in \mathcal{A}_{BGL},\ G \models_{I \to J} A \iff G' \models_{I \to J} A \\
P =_{PGL} P' & \stackrel{def}{=} & \forall A \in \mathcal{A}_{PGL},\ P \models_{m \to n} A \iff P' \models_{m \to n} A \\
W =_{LGL} W' & \stackrel{def}{=} & \forall A \in \mathcal{A}_{LGL},\ W \models_{X \to Y} A \iff W' \models_{X \to Y} A \\
D =_{DGL} D' & \stackrel{def}{=} & \forall A \in \mathcal{A}_{DGL},\ D \models_{I \to J} A \iff D' \models_{I \to J} A
\end{array}
$$

Finally we report the inverse encodings in Trees and Context Trees.

Definition A.7 (Inverse Encoding for Trees). *We consider the normal form for prime ground place graphs and we define:*

$$
\begin{array}{c}
(\![\, merge_0 \,]\!) \stackrel{def}{=} 0 \\
(\![\, K(a) \circ q \,]\!) \stackrel{def}{=} a[\, (\![\, q \,]\!) \,] \\
(\![\, merge_s \circ (M_0 \otimes \ldots \otimes M_{s-1}) \,]\!) \stackrel{def}{=} (\![\, M_0 \,]\!) \mid \ldots \mid (\![\, M_{s-1} \,]\!)
\end{array}
$$

Definition A.8 (Inverse Encoding for Context Trees). *We assume to have a bijective association between a_x and $K(a)_x$. We consider the normal of Corollary A.5 and we define:*

$$
\begin{array}{c}
(\![\, merge_0 \,]\!) \stackrel{def}{=} 0 \\
(\![\, (K(a)_x \otimes id_Y) \circ q \,]\!) \stackrel{def}{=} a_x[\, (\![\, q \,]\!) \,] \\
(\![\, (merge_k \otimes id_Y) \circ (M_1 \otimes \ldots \otimes M_k) \,]\!) \stackrel{def}{=} (\![\, M_1 \,]\!) * \ldots * (\![\, M_k \,]\!)
\end{array}
$$

$$
\begin{array}{c}
(\![\, id_1 \,]\!) \stackrel{def}{=} - \\
(\![\, (K(a)_x \otimes id_Y) \circ Q \,]\!) \stackrel{def}{=} a_x[(\![\, Q \,]\!)] \\
(\![\, (merge_k \otimes id_Y) \circ (R \otimes M_1 \otimes \ldots \otimes M_{k-1}) \,]\!) \stackrel{def}{=} (\![\, R \,]\!) * (\![\, M_1 \,]\!) * \ldots * (\![\, M_{k-1} \,]\!)
\end{array}
$$

Bigraphical Logics for XML

Giovanni Conforti
Università di Pisa

Damiano Macedonio
Università Ca' Foscari di Venezia

Vladimiro Sassone
University of Sussex

Abstract

Bigraphs are emerging as an interesting model that can represent both the pi-calculus and the ambient calculus. Bigraphs are built orthogonally on two structures: a hierarchical 'place' graph for locations and a 'link' (hyper-)graph for connections. In a previous work (submitted elsewhere and yet unpublished), we introduced a logic for bigraphical structures as a natural composition of a place graph logic and a link graph logic. Here we show that fragments of BiLogic can be used to describe XML data (with `ID` and `IDREF`s) and to reason about programs that manipulate tree-structures with query-oriented update operators.

1 Introduction

The term 'spatial,' as opposed to 'temporal,' has been recently used to refer to logics providing modal operators to express properties of the structure of the model. Such logics are usually equipped with a separation/composition operator that *splits* the current model into two parts, in order to 'talk' about them separately. Looking closely, we observe that notion of *separation* is interpreted differently in different logics.

- In 'separation' logics [23, 21], the separation is used to reason about dynamic update of tree-like structures, and it is *strong* in that it forces names of resources and pointers in separated components to be disjoint. In addition, this constraint usually implies that model composition must partially defined.
- In static spatial logics (e.g. for trees [4], graphs [7] or trees with hidden names [8]), the separation/composition operator is *structural*, and it is used to describe properties of the underlying structure. In this case no constraint on the model is usually required, and *names* may be shared between separated parts.
- In dynamic spatial logics (e.g. for ambients [10] or π-calculus [3]), the separation is intended only for location in space, and names can be shared between separated resources.

Research partially supported by '**MyThS**: Models and Types for Security in Mobile Distributed Systems', EU FET-GC IST-2001-32617 and by '**DisCo**: Semantic Foundations of Distributed Computation', EU IHP 'Marie Curie' HPMT-CT-2001-00290.

Context tree logic, recently introduced in [5], integrates the first approach above with spatial logics for trees. The resulting logic is able to express properties of tree-shaped structures (and contexts) with pointers, and it is used as an assertion language for Hoare-style program specifications in a tree memory model.

Bigraphs are an emerging model for structures in global computing, which can be instantiated to model several well-known examples, including the π-calculus, the ambient calculus and Petri nets [19]. Bigraphs consist essentially of a tree-structured place graph – representing the ambient-like nesting of locations – coupled with a link (hyper-)graph on the same nodes – representing the π-calculus-like communication channels. Such structure has recently been axiomatized in categorical terms and by means of a term algebra in [20]. In [16], we build on such bi-structural nature to introduce a '*contextual spatial logic*' for bigraphs built on two orthogonal sublogics:

- a *place graph logic*, to express properties of resource locations;
- a *link graph logic*, to express connections between resources (or, more precisely, resource names).

For this reason, we name the formalism *Bilogic*. Bilogic is interesting for at least two reasons. Firstly, the place graph logic is a generalization of both *spatial logics* and *context-tree* logic. Secondly, it captures at the same time both the notion of structural and of strong separation. In particular, we are able to express a notion of *separation-upto* that specifies which names must (or should) be shared. In addition, we are currently undertaking to extend Bilogic with temporal modalities, so as to model bigraphical reactive systems (BRS) and therefore generalize dynamic spatial logics (as, e.g., the Ambient Logic). Thus, Bilogic and its subcalculi are very general and promising as logics to talk about resources with names.

XML data are essentially tree-shaped resources, and have been modelled with unordered labelled tree in [6] where an important connection between semistructured data and mobile ambients was uncovered. Starting from *loc. cit.*, several works on spatial logic for semistructured data and XML have been proposed (e.g. [7, 17, 8]). Among these, a query language on semistructured data based on Ambient Logic was studied in [9] and implemented in [12, 15]. The present paper enriches over such model of tree-shaped data by adding links on resource names, so as to obtain a more general model for semistructured data and XML (which, as a matter of fact, it is closer to the standard OEM model). A similar step was taken in [11], which we improve upon by making use of the well-studied categorical structure of bigraph, which internalize the notion of link and makes the difference between strong and structural separation explicit. In addition, bigraphs naturally model XML contexts: we thus obtain with no additional effort a logic to describe XML contexts which can be interpreted as web services or XML transformations.

Here we focus on the applications of bigraphical logics to XML data. In particular, we first show how XML data (and, more generally, contexts or positive web services) can be interpreted as a bigraph. Equipped with such 'bigraphical' representation of XML data and contexts, we then give a gentle introduction to different fragments of Bilogic and show how they can be applied to describe and reason about XML. The

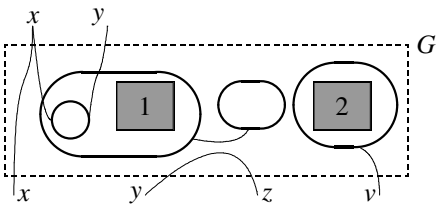


Figure 1: Bigraph G

contribution of the paper is therefore to identify (fragments of) Bilogic as a suitable formalism for semistructured data, and illustrate its expressiveness by means of selected examples.

Structure of the paper. In §2 we recall the basic notions of bigraphs; for a fuller exposition the reader is referred to [19]; §3 shows how to interpret XML contexts as bigraphs, while §4 illustrates several examples.

2 Bigraphs

We give a crush introduction to bigraphs [19]. We restrict our attention exclusively to *abstract pure bigraph*, whose algebraic axiomatization has been provided in [20].

The bigraph G in Fig.1 is a structure built on two orthogonal graphs over the same group of nodes, shown here with bold outlines. Nodes may be nested in a hierarchical tree structure (the *place graph*, shown as the inclusion of a node in another), and have *ports* that may be connected by *links* (the *link graph*, shown as edges connecting nodes). The nesting of nodes imposes no constrain upon the linkage of their ports, hence the orthogonality of the structures.

Every node of the place graph has a *control*, which represent what kind of node it is and its *arity*, i.e., an ordinal telling the number of ports and marking them unambiguously. In the Figures 1 and 2, arity is one for oval shaped nodes, two for the round ones, three for the triangular one.

At the top level of the nesting structure are the *roots*. in the figure there is a unique root, shown as a dotted outline. In general however there may be multiple roots. Inside a node there may be *holes*, shown as shaded boxes in the figure, which formalize *contexts*. Roughly speaking, a place graph is a list of (unordered) trees that can have *holes* as leaves. Place graphs are characterized by a couple of ordinals, written as $m \to n$, denoting respectively the number of holes and the number of roots in the bigraph. The use of ordinals allows us to mark holes and roots uniquely.

The link graph is characterized by a couple of set of names $X \to Y$. The set X represents the *inner names* (drawn in the figure below the bigraph) and Y represent the set of *outer names* (drawn above the bigraph). The link graph connects a port to a name or an *edge*. In Fig. 1 an edge is represented by a line between nodes. Ports

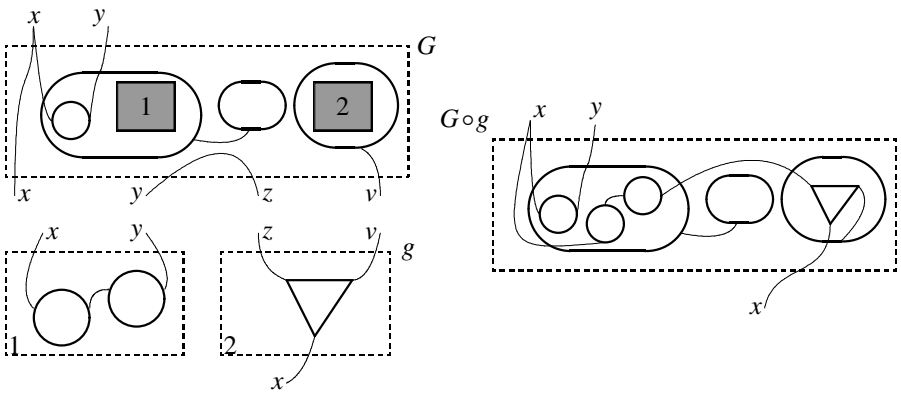


Figure 2: Bigraphical composition

can be associated both to names and to edge, in any finite number. A link to a name is *open* and may be connected to other nodes in case of composition between bigraphs. A link to an edge is *closed* and cannot be connected to other ports. The role of names becomes clearer after introducing the composition operator of bigraphs.

The right hand side of Fig. 2 represents the *composition* $G \circ g$ between the two bigraphs G and g described on the left hand side. The main idea is to insert g into the context G. The operation is partially defined, since it requires the inner names and the number of holes of G to match to the number of roots and the outer names of g respectively. Shared names create the new links between the two structures. Intuitively, composition *first* places root region of g in the proper hole of G and *then* joins equal inner names of G and outer names of g. In particular note the edge connecting the inner names y,z in G, its presence produces a link between two internal nodes of g after the composition.

As explained in [19], the tensor product $\otimes$ of bigraphs is defined only if they do not have common inner names or outer names. By relating names to resources, the tensor can be seen a separation operator: the product puts the place graphs one next to the other (in order), obtaining a graph with more roots and holes, and operates the disjoint union between inner names and outer names of the place graphs.

3 Modelling XML Contexts as Bigraphs

As proved in [20], the class of bigraphs can be axiomatized using a small set of elements. We recall the constructions below, and then relate it to XML. In our formalization, XML data are bigraphs with no holes (i.e., *ground*), while those with holes represent XML contexts.

The main constituent of a bigraph is the *discrete ion* $K_{\vec{a}}$, which represents a node with one root and containing one hole. The hole can be filled with other ions in order

to build a more complex tree-structure. The ion's control is K, with arity $ar(K) = |\vec{a}|$, and every port of $K_{\vec{a}}$ is linked to a name in the (ordered) list of names $\vec{a}$. Every name in $\vec{a}$ represent an outer name of the bigraph. Thinking in terms of XML data, a ion is seen as a *tag* with some *attributes*. Since arity is an ordinal, it is possible to identify the ports unambiguously and it is easy to associate them to attributes. We assume one designed port to be associated to a (unique) name used to identify the element, as an `ID` attribute. Other ports may be linked to other nodes' `ID` names, so acting effectively as `IDREF`s, or to internal edges connected to internal nodes, representing the general attributes of the element. Embedding a ion into the hole of another ion, represents the inclusion of the corresponding elements.

The basic place graphs are 1, id_n and *merge*. Term 1 is the empty single rooted bigraph, it is the empty XML document; id_n is a context with n holes, n roots and no internal node. It behaves like a neutral XML context if composed with a compatible XML document. Term *merge* achieves the merging of two bigraphs: it consists of two holes, one root and no links. By composing merge with the product of two single-rooted XML documents we obtain a XML document whose single root has the two component documents as children.

The basic components of a link graph are $(a \leftarrow b)$, $(a \leftleftarrows b)$ and $/a$. Operator $(a \leftarrow b)$ represents renaming, and in the context of our interpretation of XML, it acts on `ID` attributes. Operator $(a \leftleftarrows b)$ associates name b to name a: when a represent an `ID` and b a `IDREF`, then $(a \leftleftarrows b)$ makes b a reference to a. Finally, $/a$ makes name a private, and allows nodes to be joined to one another in closed links. This operator will be used in our encoding of XML to express a link between attributes and their values.

In the presence of (unfilled) holes, terms represent *contexts* for XML data, i.e., documents with holes to be filled by XML data: composition acts by inserting documents in such holes. The tensor is defined only if the names appearing in the two components are disjoint. Therefore, any reference 'going across' must be created after the product. Note that since link graphs in general perform substitution and renaming, the outer names of g may not be outer names of $G \circ g$. This may happen either because they are renamed or because an edge has been added to the structure as effect of the composition, which makes the link *private*, i.e., without an externally-visible name.

The parent-child relationship on nodes in bigraphs does not capture order among children of the same node. So bigraphs can be seen as a (ordered) list of unordered (contexts of) trees connected through links. This model can be used for XML data whose document order is not relevant. Such documents arise for instance in XML encodings of relational databases [2], in the integration of semi-structured database sources, or in the case of distributed XML documents in a P2P environment [22] (in the latter case the document order is not defined).

The importance of the underlying hierarchical structure in XML, and the fact that links are used sporadically only for modelling relations between nodes, suggests the bigraphical model as a good model for XML documents. We interpret these documents as ground bigraphs by using the encoding explained below. The encoding is trivial in case the tree contains no attribute, when we can in fact easily map the tree structure of XML elements into the place graph by associating controls to tags and values. In this case, there is no link between nodes, all controls have arity zero, and the XML file is completely modelled by the place graph only (in a kind of ambient like formalism [6]).

In the case of elements with attributes, we need names to represent XML links between elements (e.g., like `ID-IDREF` relationships), and edges to represent elements' attributes. We consider the `ID`s used in XML data as names in bigraphs. The encoding is defined by assuming two functions on values:

- $K_{val}(v)$, mapping the value v to a ground bigraph corresponding to a single rooted node with no outer names, no nodes and no holes inside.
- $K_{val}(v)_a$, mapping the value v to a ground bigraph corresponding to a single rooted node with outer name a, no nodes and no holes inside.

The former function is used actually to encode values with bigraphs, the latter is auxiliary and encodes values linked to attributes.

Moreover we associate tags with ions. We assume a class $\mathcal{K}_{tag}$ of controls. We consider a tag t and firs we observe that the list Att of its attributes is finite and ordered, hence we associate the list to an ordinal $\#Att$, and the elements of the list are identified by their position. Then we associate t with $K_{tag}(t, \#Att)_{\vec{u}}$, that is a ion with control $K_{tag}(t, \#Att) \in \mathcal{K}_{tag}$ and arity $\#Att$. The vector $\vec{u}$ indicates the names connected to the control; we assume the names in $\vec{u}$ to be the `ID`s associated to the attributes in Att.

A value attribute is encoded as a value inside the node and connected to the port whose position marks the corresponding attribute. Identifiers (like `ID`) and links (like `IDREF`) attributes have a special interpretation. They become *names* of the tag and can be connected with other names in order to model references. As mentioned before, the connection is performed by using the link graphs constructors: $(a \leftleftarrows b)$, to create a reference, and $/a$, to create a closed connection for attributes. The general definition for the encoding is formalized in Tab. 1

Table 3.1. *XML documents as ground bigraphs*

$(\!\| v \|\!)$	$\stackrel{def}{=} K_{val}(v)$	value
$(\!\| v \|\!)_a$	$\stackrel{def}{=} K_{val}(v)_a$	value linked to an attribute name a
$(\!\| \vec{v} \|\!)_{\vec{b}}$	$\stackrel{def}{=} (\!\| v_1 \|\!)_{b_1} \otimes \ldots \otimes (\!\| v_n \|\!)_{b_n}$	with $\vec{v} = v_1 \ldots v_n$ and $\vec{b} = b_1 \ldots b_n$
$(\!\| \emptyset \|\!)$	$\stackrel{def}{=} 1$	empty tree
$(\!\| T \|\!)$	$\stackrel{def}{=} /\vec{a} \circ \sigma \circ K_{tag}(t, k+p+1)_{u,\vec{u},\vec{b}} \circ merge_{n+k}((\!\| \vec{v} \|\!)_{\vec{b}} \otimes \alpha_1 \circ (\!\| T_1 \|\!) \otimes \ldots \otimes \alpha_n \circ (\!\| T_n \|\!))$	
whit	$T = \langle t,\ \texttt{ID} = u,\ \vec{a} = \vec{u},\ \vec{b} = \vec{v} \rangle\, T_1, ..., T_n \,\langle /t \rangle$	XML tree
	$\vec{a} = a_1 \ldots a_k$	link attributes
	$\vec{u} = u_1 \ldots u_k$	names
	$\vec{b} = b_1 \ldots b_p$	value attributes
	$\vec{v} = v_1 \ldots v_k$	values
	α_i	renaming the names of T_i into fresh names
	$\sigma = \alpha_1^{-1} \cup \ldots \cup \alpha_n^{-1}$	inverse renaming
	$/\vec{a} \stackrel{def}{=} /a_1 \otimes \ldots \otimes /a_p$	closure of the names in $\vec{a}$
	$merge_{n+k}$	merging among $n+k$ bigraphs (definable from *merge*)

In the table above, the encoding of values is simply the function $K_{val}(\,)$. The auxiliary encoding of values linked to attributes is given by $K_{val}(\,)_a$. Term 1 corresponds to the empty tree. The core of the translation is the encoding of (non empty) trees. Here, the role of *merge* is to group together the (encodings of the) set of children of T and the (encodings of the) values linked to attributes. In this case, values linked to attributes are associate with a name. Observe in the encoding the use the renamings α_i to guarantee the product is defined, since it requires the names to be distinct. We choose fresh names, i.e., not appearing in T, and we obtain the renamings α_i by combining different operators such as $(a \leftarrow b)$. The obtained bigraph is single rooted, hence it fits in the ion associated to the tag t. After the composition with the ion, we have to rename the names in order to formalize all the references, finally we need to close the link between the root and the evaluations of the values linked to attributes. The renaming is obtaining by considering the inverse of α_i (definable by using operators such as $(a \leftarrow b)$ and $(a \leftleftarrows b)$), and the closure is obtained by combining the closure of every name associated to an attribute.

Example. As an example of the encoding of XML trees into bigraphs, we consider a database that stores scientific papers and information about their authors. We focus on the fragment quoted in the document below.

```
<authors>
  <author name="Conf" n="ID2" coauth="ID5">
    <Address n="ID1">"."</Address>
    <Phon n="ID3">"."</Phon>
  </author>
  <author name="Sass" n="ID5" coauth="ID7">
    <Address n="ID4">"."</Address>
    <Phon n="ID6">"."</Phon>
  </author>
  <author name="Mace" n=ID7>
    <Address n="ID8">"."</Address>
    <Phon n="ID10">"."</Phon>
  </author>
</authors>
```

Tag *Author* has an identifier, $\texttt{ID}_i$, a link to another author, `coauth`, that is an `IDREF` attribute, and a general attribute, `name`. The encoding is illustrated in Fig. 3. Every tag *Author* is associated to a control of arity three. Exploiting the order of the ports, we identify a port with the corresponding XML attribute unambiguously. In the picture we assume the ports ordered clockwise. The first port corresponds to the general attribute `name`, and is connected by a close link (an edge) to a value. The second one corresponds to the identifier, `ID`, and is connected to an outer name. The final attribute corresponds to the reference , `coauth`, and is connected to a name that correspond to another *Author* tag.

More generally a bigraph can be seen as a context for unordered XML data, just because there can be holes in it. So in the previous example we can imagine to put holes in place of some node. This yields a context that can be interpreted as a contextual

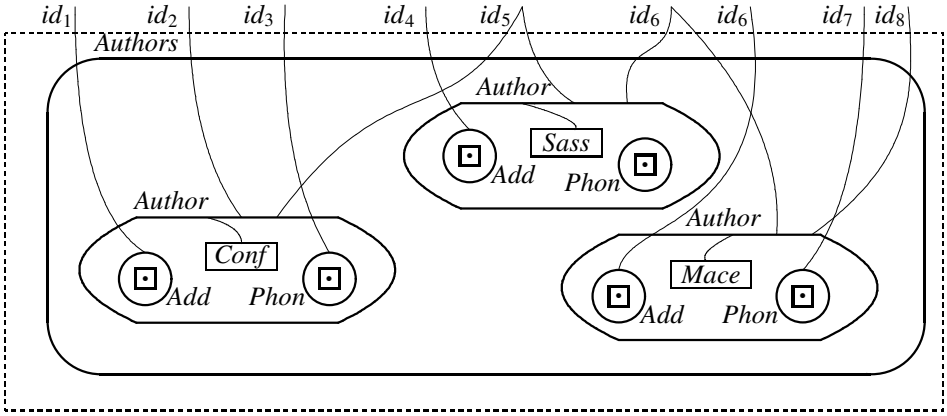


Figure 3: XML encoding

XML document, a function (*Web service*) that takes a list of XML files and returns their composition in context, by fitting every file in the relative position (as marked by its position on the list). In this way we can model Web services, besides plain XML documents. In order to model the order of XML elements, we would need to add a notion of ordered locality, i.e., to consider molecules with an ordered "list" of holes, and extend the theory of bigraphs accordingly to this notion.

4 Bilogics for XML Contexts

In [16] we introduced Bilogic for bigraphical structures. In §3 we have shown that XML (unordered) data and contexts can be modelled as a bigraphical structures. This section briefly introduces Bilogic, and explains how it can be used for describing, querying and reasoning about XML. In particular, we analyze three possible cases:

- Logics for place graphs to model XML data and contexts without `IDs`;
- Logics for *discrete bigraphs* (essentially trees with unique identifiers) for XML with `IDs`, but without links;
- Bigraphical logics for XML with `IDs` and links.

4.1 XML without `IDs`

As mentioned previously, without attributes – or anyway handling them as children elements themselves – XML amounts to unordered labelled tree. In [6] the author shows that such a model has some similarities with ambient calculus terms, building on which [9] introduces a query language for semistructured data based on Ambient Logic. In [16] we show that the static fragment of ambient logic (STL) can be easily

extended to the Place Graph Logic (PGL in the following) to model general contexts of tree-shaped resources. In particular PGL can describe place graphs, that is bigraphs without links, and so it can be used to talk about XML contexts (without attributes) using the encoding we defined in the previous section. We briefly present here some operators of PGL, and we informally show their semantics in the XML case. (Some of connectives are derived; the reader is referred to [16] for the details.)

Table 4.1. *PGL: Place Graph Logic (some operators)*

$A, B ::=$	formulas
$\mathbf{F}$	false
$A \Rightarrow B$	implication
$\mathbf{1}$	empty single rooted bigraph
$\mathbf{id}_n$	identity on n number of holes (even zero)
$A \otimes B$	decomposing in two place graphs one next to the other
$A \circ_s B$	possible s-holes filling of A with a place graphs satisfying B
$A \circ\!\!-_s B$	if inserted in a context A (with s holes) then B
$K[A]$	the molecule K containig something satisfying A
$A \mid B$	decomposing in two trees whose merge is the current model

$\mathbf{F}$ and $A \Rightarrow B$ are the standard propositional operators (the other propositional connectives $\mathbf{T}$, $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$ are derived as usual). There are spatial constants $\mathbf{1}$, **merge**, and id_n denoting a singleton place graph (interpreted as a single XML context). We interpret $\mathbf{1}$ to be the empty XML context, **merge** the context merging two XML contexts in one, while **id** is the identity context, which transforms XML trees to themselves. The two spatial operators $A \otimes B$ and $A \circ_s B$ express two ways of composing contexts. The first is *horizontal* and produces a (ordered and separated) pair of contexts one next to the other. The second one is *vertical* and corresponds to fill the s holes of a context satisfying A with the context satisfying B. They are both non commutative. Then, there are ambient-like operators for trees: $K[A]$ is the context that inserts a new root labelled K in the top of a single XML context satisfied by A, and $A \mid B$ (parallel composition) denotes contexts obtained by merging the tree contexts satisfying A and B in a single root. Note that, since parallel composition performs a merge of the contexts, it provides a commutative monoid with $\mathbf{1}$ as neutral element. An interesting connective is $A \circ\!\!-_s B$, which essentially expresses that whenever the current model is inserted inside a XML s-ary context satisfying A, then the resulting context satisfies B.

In general, models of PGL are *positive* functions from m to n that given a list of m XML contexts produces a list of n XML contexts. By 'positive' we mean that they can only add structure to the parameters, and not remove or replace parts of them. In this sense, XML contexts are viewed as positive XML Web services that take XML documents (possibly with calls to other Web services, so that they effectively are XML contexts), and return XML documents. This is similar to the model of Positive Active XML proposed in [1], but with a remarkable difference: since our model does not handle ordered trees, we cannot restrict attention to functions between XML (active) documents. We need to use with *list* of parameters and a *list* of resulting contexts. To

understand better the idea, consider the Web service below.

$$wb : K_1[id_1] \mid K_2[merge \circ id_2]$$

It takes three trees and puts the first inside a node labelled K_1, merges the second and third trees and puts the result inside a node labelled K_2, and finally produces the parallel composition of the two resulting trees. We need ordered parameters to put the right root in the right hole. A Web service like this can be solely identified by a characteristic formula (corresponding to the tree), but more generally a formula like $K_1[id_1] \mid \mathbf{T}$ can match all Web services having at least one hole and decomposable as a node of arity one labelled K_1 in parallel to something else. In this sense a notion of *type* for Web services arises. Similarly to [13], where the spatial tree logic is used to describe XML types and constraints, we can use PGL to formalize Web service types and constraints.

Since also XML (active) documents are contexts, we can actually use the PGL to describe Active XML documents and Web service in an unique framework. In addition, we can use an approach like TQL [9] to query Active XML documents and Web service, and eventually use types to avoid Web service useless invocations. To make the idea more precise, with reference to the previous example, take a query $wb \circ (T_1 \otimes T_2 \otimes T_3) \models K_1[X] \mid \mathbf{T}$, which essentially determines all contexts reachable from the result of the Web service invocation through a path $/K_1$. If we know that the type of wb is $K_1[\mathbf{id}_1] \mid K_2[\mathbf{T}]$ we can avoid to evaluate the web service by observing that $(K_1[\mathbf{id}_1] \mid K_2[\mathbf{T}]) \circ\!\!-\!\!\!\!\!\!\rightarrow_3 K_1[X] \mid \mathbf{T}$, and so:

$$wb \circ (T_1 \otimes T_2 \otimes T_3) \models K_1[X] \mid K_2[\mathbf{T}] \quad \Longleftrightarrow \quad T_1 \models \mathbf{id} \circ X.$$

4.2 XML Contexts with `ID`

In the previous section we focused on the place structure only. Since logic and model have no way to directly identify resources, it is only possible to access a resource through navigation. A different approach is possible when the XML document has identifiers for and pointers to elements. In this case, the tree model can be seen as an extension of a heap memory model in which locations are referred to by names. Such names are intrinsically separated by the tensor product, which is defined only on structures which disjoint name sets. We can see such models as discrete bigraphs, i.e., place graphs with named resources but no name sharing between different resources. A logic for these is introduced in [16] as an extension of the PGL with named (identified) controls K_x and renamings $(x \leftarrow y)$. Such a logic is able to express properties of (contexts of) resources that can be accessed in two ways: as usual, by navigation through the tree structure, and by using names controls as pointers.

The logic essentially adds two operators to PGL:

$$\begin{array}{rl} K_{\vec{a}} & \text{for named nodes;} \\ (a \leftarrow b) & \text{for renaming these names.} \end{array}$$

The ion $K_{\vec{a}}$ has a list of names, although in the case of XML with identifiers and no links only one name is needed. Thus, we write K_x to denote the node (with an hole) inside labelled K with name identifier x, and the formula K_x denotes this XML context

only. The rename ($a \leftarrow b$) is needed in order to map names of different sources to different identifiers (e.g., $(x \leftarrow y) \circ K_y = K_x$). The tensor product now constraints the models to be separated both in locality and in names, i.e., when we write $A \otimes B$ we mean that the models satisfying A and B have disjoint sets of identifiers (that is disjoint outer faces). On the other hand the composition $A \circ_s B$ is defined when the inner face of A and outer face of B coincide.

With this logic we have two ways to address separation: by means of the tensor product, or by using the composition. For example we can think to encode a heap-like structure as $(x_1 \mapsto a_1) \# ... \# (x_n \mapsto a_n)$ in two ways:

- *horizontally*, as a one level forest $K(a_1)_{x_1} \circ 1 \otimes \ldots \otimes K(a_n)_{x_n} \circ 1$;
- *vertically*, as a line $K(a_1)_{x_1} \circ \ldots \circ K(a_n) x_n \circ 1$. (This matches with [5], where separation logic can be expressed using composition only.)

where $K(a_i)_{x_i}$ is a ion associated to an a_i, and 1 fills the holes.

While the first approach can be lifted to a commutative monoid if we use the merge operation, the latter cannot be interpreted in a commutative way directly. In this sense discrete bigraphs are more general than heaps of the form above (as used in [23, 21]). We believe we can extend the result of [16], viz. the encoding of Context Tree Logics on unidentified nodes into PGL, to Discrete Bigraphs Logics, so as to model the entire Context Tree Logic of [5], including identifiers. Thus, this logic can also be used to reason about programs in tree-shaped memory models.

4.3 XML Contexts with Links

In order to model sharing of names between resources and treat structures with pointers, we have to extend the logic of discrete bigraph with a notion of sharing. The sharing is obtained in bigraphs through links between names of resources. In our case, we have encoded identifiers as tag names and `IDREFs` as pointers to names in the same document. In [16] we have introduced a logic for general bigraphs as a composition of a link graph logic and a place graph spatial logic. Such a combination is very expressive, and induces a hiding operator for local/private/hidden names. For the present application to XML this is only needed for the encoding of value attributes. On the other hand, we require a notion of separating conjunction with sharing, in order to express properties like: "*The author of paper X has a relationship with the author of paper Y.*" In fact, this property expresses separation on resources (different authors of different papers), but sharing on linked names. Such operator is explicitly introduced in [16] by using the tensor product of Bilogic, the renaming function and the *freshness* operator of nominal logics. The main idea is that a link between names can be seen as a separation between separated names that are then linked by means of substitution. In [14, 8] a notion of link similar to this is hidden in the model because the ambient-like operation shares names by default (which may be the main reason for undecidability of logics with abstract names).

5 Conclusions

In the paper we have sketched the application of Bilogic to describe and reason about XML data. This is however not the main reason why Bilogic and bigraphs are interesting for XML and other global resources in general. Bigraphs were introduced basically to model dynamic concurrent systems (cf. [18]), where they are used as a contextual way to specificy reaction rules. We believe that Bilogic, inheriting the flexibility and universality of such model, will help creating a general logic framework uniformly applicable to several actual calculi. The study of the case of XML was initiated here, and in the future we plan to extending to more sophisticated semistructured data models.

Acknowledgment We would like to thank Philippe Bidinger, Robin Milner and Peter O'Hearn for useful discussions. A special thank to Carlo Sartiani.

References

[1] S. Abiteboul, O. Benjelloun, and T.Milo. Positive active xml. In *Proc. of PODS 2004*, 2004.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from relations to semistructured data*. Morgan Kaufmann, 1999.

[3] L. Caires and L. Cardelli. A spatial logic for concurrency (Part I). In *Proc. of Theoretical Aspects of Computer Software; 4th International Symposium, TACS 2001*, volume 2215 of *LNCS*, pages 1–37. Springer-Verlag, 2001.

[4] C. Calcagno, L. Cardelli, and A. D. Gordon. Deciding validity in a spatial logic for trees. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, 2003.

[5] C. Calcagno, P. Gardner, and U. Zarfaty. A context logic for tree update. In Proc. of LRPP 2004, revised version to appear in POPL 2005.

[6] L. Cardelli. Describing semistructured data. *SIGMOD Record, Database Principles Column*, 30(4), 2001.

[7] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *Proc. of ICALP*, volume 2380 of *LNCS*, page 597. Springer-Verlag, 2002.

[8] L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *Proc. of FOSSACS '03*, volume 2620 of *LNCS*, pages 216–232. Springer-Verlag, 2003.

[9] L. Cardelli and G. Ghelli. Tql: A query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14:285–327, 2004.

[10] L. Cardelli and A. D. Gordon. Ambient logic. To appear in Mathematical Structures in Computer Science.

[11] L. Cardelli, P.Gardner, and G. Ghelli. Querying trees with pointers. Unpublished notes.

[12] G. Conforti, O. Ferrara, and G. Ghelli. TQL Algebra and its Implementation (Extended Abstract). In *Proc. of IFIP TCS*, pages 422–434. Kluwer Academic Publishers, 2002.

[13] G. Conforti and G. Ghelli. Spatial logics to reason about semistructured data. In *Proc. of SEBD 2003: Eleventh Italian Symposium on Advanced Database Systems*. Rubettino Editore, 2003.

[14] G. Conforti and G. Ghelli. Decidability of freshness, undecidability of revelation. In *Foundations of Software Science and Computation Structures, FOSSACS 2004*, pages 105–120. Springer, 2004.

[15] G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The Query Language TQL. In *Proc. of 5th International Workshop on Web and Databases (WebDB 2002)*, 2002.

[16] G. Conforti, D. Macedonio, and V. Sassone. Bilogics: Spatial-nominal logics for bigraphs (extended abstract). Submitted for publication. Available from `http://www.di.unipi.it/~confor/publications.html`, October 2004.

[17] Silvano Dal Zilio and Denis Lugiez. A logic you can count on. In *POPL 2004 – 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[18] O. H. Jensen and R. Milner. Bigraphs and transitions. In *Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–49. ACM Press, 2003.

[19] O. H. Jensen and R. Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580. University of Cambridge, February 2004.

[20] R. Milner. Axioms for bigraphical structure. Technical Report UCAM-CL-TR-581. University of Cambridge, February 2004.

[21] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *In Proc. of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer-Verlag, 2001.

[22] Sigmod record volume 3 number 1, 2004. special topic section on peer to peer data management, 2004.

[23] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74. IEEE Computer Society, 2002.

Formal analysis of multi-party contract signing

Rohit Chadha*
University of Sussex
rchadha@sussex.ac.uk

Steve Kremer†
Université Libre de Bruxelles
skremer@ulb.ac.be

Andre Scedrov ‡
University of Pennsylvania
scedrov@saul.cis.upenn.edu

Abstract

We analyze the multi-party contract-signing protocols of Garay and MacKenzie (GM) and of Baum and Waidner (BW). We use a finite-state tool, MOCHA*, which allows specification of protocol properties in a branching-time temporal logic with game semantics. While our analysis does not reveal any errors in the BW protocol, in the GM protocol we discover serious problems with fairness for four signers and an oversight regarding abuse-freeness for three signers. We propose a complete revision of the GM subprotocols in order to restore fairness.*

1. Introduction

The problem of digitally signing a contract over a network is more complicated than signing a contract by "pen and paper". The problem arises because of an inherent asymmetry: no signer wants to be the first one to sign the contract because another signer could refuse to do so after having obtained the first signer's contract.

A simple solution consists in using a trusted party (T) as an intermediary. Signers send their respective contracts to T, which first collects the contracts and then distributes them among the signers. An intermediary is known to be necessary [10]. However, because of the communication and computation bottleneck at T, this solution is inefficient. Other solutions include randomized protocols as well as protocols based on gradual information exchange. More recently, the so-called *optimistic* approach was introduced in [2, 5]. The idea is that T intervenes only when a problem arises, *e.g.,* a signer is trying to cheat or a network failure occurs at a crucial moment during the protocol. Such protocols generally consist of a main protocol and one or several subprotocols, each with a fixed number of messages. The main protocol is executed by the signers in order to exchange their signatures. The subprotocols are used to contact T in order to force a successful outcome or to abort the protocol.

A contract-signing protocol should respect several desirable properties. The first property is *fairness*. Intuitively, a contract-signing protocol is fair if at the end of the protocol either each signer obtains all the other signers' contracts or no signer gets any valuable information. A second property, *timeliness*, ensures that signer has some recourse to prevent endless waiting. Both fairness and timeliness are standard properties that are also important in fair exchange, certified e-mail and fair non-repudiation protocols. A property that is specific to contract signing, *abuse-freeness*, was introduced in [11]. A protocol is abuse-free if no signer A is able to prove to an external observer that A has the power to choose between successfully concluding the protocol and aborting the protocol. A protocol that is not abuse-free gives an undesirable advantage to one signer, say Alice, who has the power to decide the outcome of the protocol and can prove this to an external observer. If, for instance, Alice wants to sell a house to Charlie, she could initiate a contract with Bob just to force Charlie to increase his offer.

There have been several applications of formal methods to contract signing, so far only for the special case of two signers. The finite model-checker Murφ is used in [16] to analyze two contract-signing protocols, discover subtle errors and suggest corrections. In [6] inductive methods are used to reason about contract-signing protocols specified in the multiset-rewriting framework, MSR. Protocol properties are expressed in terms of strategies, which provide a natural framework for the analysis. In [15] the finite model-checker MOCHA is used to analyze two contract-signing protocols. The advantage of using MOCHA rather than Murφ is that MOCHA allows to specify protocol properties in ATL, a temporal logic with game semantics, which in turn allows reasoning about strategies. In [13] the finite state tool SHVT is used to analyze several variants of the Zhou-

* Partially supported by the ONR CIP/SW URI "Software Quality and Infrastructure Protection for Diffuse Computing" through ONR Grant N00014-01-1-0795 and by the NSF Grant CCR-0098096 and the EU Global Computing project "MIKADO".

† This research was carried out while the author stayed at University of Pennsylvania funded by the "Communauté Française de Belgique".

‡ Partially supported by the ONR CIP/SW URI "Software Quality and Infrastructure Protection for Diffuse Computing" through ONR Grant N00014-01-1-0795 and by the NSF Grant CCR-0098096.

Gollmann non-repudiation protocols (non-repudiation protocols are closely related to contract signing protocols). Protocols are modeled using asynchronous product automata and properties are basically invariants. They show unknown attacks which can occur in a realistic implementation of the protocol. The most recent work on contract signing [8] introduces the notion of an *optimistic* signer, *i.e.,* a signer that prefers to wait for "some time" for messages from the other signers before contacting the trusted party. The main theorem of [8] is that, independently of a specific protocol, if any of the signers is optimistic, then the other signer will at some point of the protocol have the power to decide the outcome.

All the efforts just described consider only two-party protocols. In this paper we analyze multi-party contract-signing protocols [4, 12]. The protocol goal in that case is that each signer sends its signature on a previously agreed upon contract text to all other signers and that each signer receives all other signers' signatures on this contract. In a multi-party framework, fairness, timeliness, and abuse-freeness should hold against *any* coalition of dishonest parties. Unlike in the two-party case, the complexity level of the multi-party protocols, especially [12], is such that a tool, e.g., a model-checker, is indispensable in the analysis. This partly comes about from an important difference between the two-party and the multi-party case, namely, in the multi-party case T has to be able to overturn its previous abort decisions [11]. As our analysis shows, this feature is particularly difficult to design correctly. We have discovered an essential obstacle in the GM protocol [12], which appears not to be removable without completely changing the sub-protocols for T and which leads to the failure of fairness in the case of four signers. We present this attack in detail in the paper and propose a corrected version of the GM protocol, which has been validated by MOCHA. MOCHA did not find any problems with fairness in the BW protocol [4] nor in the original GM protocol with only three signers. In the latter case, MOCHA did find an amusing problem with abuse-freeness, but this problem is easily corrected. We believe that the main reason for robustness of the BW protocol is that overturning the aborts decisions has been designed correctly. We will discuss the BW protocol only briefly because of the lack of space. Note that the choice of using MOCHA rather than another model-checker was mainly influenced by the facts that we were familiar with the tool and that ATL provides an elegant way of modeling properties, in particular abuse-freeness. We suppose that similar results can be obtained using other model-checkers. A preliminary announcement of these results has been made in Workshop on Issues in the Theory of Security [7].

Outline of the paper. The rest of the paper is organized as follows. In section 2, we describe the BW and the GM protocols. In section 3, we present briefly the finite-state tool, MOCHA, the temporal logic ATL and its game semantics. Modeling of the protocols and protocol assumptions in the game semantics along with the modeling of fairness in ATL is briefly discussed. In section 4, we report on our analysis of the BW and GM protocols using MOCHA, present the fairness attack on four signers in detail and propose a corrected version version of the protocol. We discuss briefly how to restore fairness and present the anomaly with respect to abuse-freeness for three signers. In order to detect this anomaly, we had to model optimistic signers and discuss this issue. We summarize our results and discuss directions for future work in section 5.

Acknowledgments. We thank Rajeev Alur, Iliano Cervesato, Juan Garay, Philip MacKenzie, Carl Gunter, Joshua Guttman, F. Javier Thayer Fabrega, Catherine Meadows, Dusko Pavlovic and Paul Syverson for interesting and helpful discussions. We would also like to thank the anonymous referees for their useful comments.

2. Protocol description

In this section we describe the multi-party contract-signing protocols proposed Garay and MacKenzie in [12] and briefly discuss the protocol proposed by Baum and Waidner in [4]. Unlike two-party protocols, which generally have similar structures, the two multi-party protocols described below have fundamentally different structures. For this section and for the rest of the paper, we shall assume that each protocol participant has a private signing key and a corresponding public verification key. Each participant shall be identified with this private/public key pair, and if we say that "*A can* ...", we shall mean anyone that possesses the private key of A.

2.1. GM multi-party contract-signing protocol

The protocol allows n ($n \geq 2$) participants, say $P_1, \ldots, P_n$, to exchange signatures with the help of a trusted party T on a preagreed contract text m. P_i is said to have a *contract* if it has everybody's signature on the text m. The order of the participants $P_1, \ldots, P_n$, henceforth referred to as *signers*, and the identity of T are also agreed upon before the protocol begins. The preagreed contract text m contains an identifier that uniquely identifies each protocol instance. In [12], the communication amongst the participants is assumed to be over a network channel in control of a "Dolev-Yao intruder", while the communication between the participants and the trusted party is assumed to be over a private channel.

The protocol uses zero-knowledge cryptographic primitives, *private contract signatures*, that were first intro-

duced in [11]. The private contract signature of A for B on text m with respect to a trusted party T, denoted as $PCS_A(m, B, T)$ has the following properties:
a) $PCS_A(m, B, T)$ can be created by A.
b) $PCS_A(m, B, T)$ can be faked by B. Only A, B and T can tell difference between PCS and its simulation.
c) $PCS_A(m, B, T)$ can be converted into a conventional universally-verifiable digital signature, $S_A(m)$, by both A and T. Only A and T can do this conversion.

The protocol itself consists of three subprotocols: *main*, *abort*, and *recovery* subprotocols. Usually signers try to achieve the exchange by executing the main subprotocol. They contact T using one of the other two subprotocols when they think something is amiss. Once a signer contacts T, it no longer takes part in the main subprotocol. T responds to a request with either an abort token or a signed contract. The decision whether to reply with an abort token or with a signed contract is based on a database maintained by T, which stores all the relevant information of the requests and its responses. Once T sends back a signed contract, it always replies with the signed contract. As discussed below, a decision to abort may, however, be overturned in order to maintain fairness. We discuss the subprotocols in some detail.

Main protocol. The main protocol for n signers is divided into n-levels, that can be described recursively. For each level of recursion, a different "strength" of promise is used. The strength of a promise is denoted by an integer "level", and an "i-level promise from signer A to signer B on a message m" is implemented using PCS: $PCS_A((m, i), B, T)$.

In level i, signers P_i through P_1 exchange i-level promises to sign the contract. The i-level protocol is triggered when P_i receives 1-level promises from $P_{i+1}, \ldots, P_n$. After receiving these promises, P_i sends out its 1 level promise to signers $P_{i-1}, \ldots, P_1$ and waits for $i-1$ level to finish. At the end of the $i-1$ level, P_1, $\ldots P_{i-1}$ have exchanged $i-1$ level promises and P_i receives a $i-1$ level promise from each of the signers $P_1, \ldots, P_{i-1}$. Now $P_i \ldots, P_1$ exchange i level promises, and close the higher levels.

In order to close level a where $a > i$, P_i sends an $(a-1)$-level promise to P_a and waits for a-level promises from signers $P_{i+1}, \ldots, P_a$. After receiving these promises, P_i indicates its willingness to close the level a to signers $P_1, \ldots, P_{i-1}$ by sending them its a-level promise, and in return waits for a-level promises from them. Upon receiving these, P_i sends its a-level promises to $P_{i+1}, \ldots, P_a$ completing its obligation in the a-level protocol. P_i then proceeds to complete $a+1$ level.

Once the n-levels are completed, each signer has a n-level promise from everybody else, and the contract exchange is ready to begin. In this exchange, each signer also sends a $n+1$-level promise to everybody along with its signature on the preagreed text. In order to complete the exchange, signer P_i waits for the contract and $n+1$-level promises from $P_n, \ldots, P_{i+1}$. Upon receiving these, P_i sends its signature and $n+1$-level promises to everybody, and waits for the signatures and $n+1$-level promises from $P_{i-1}, \ldots, P_1$. Once these are received, the protocol ends for P_i, and P_i has the contract.

If some expected messages are not received, P_i may either quit the protocol or contact T. P_i may simply quit the protocol if it has not sent any promises or contact T if it has sent some promises. It may contact T with a request to abort if it has not received any promise from some signer. It may request T to recover the protocol if it has a promise from every other signer. A detailed description of the main protocol is given in table 1.

In order to illustrate the main protocol consider an instance of the protocol with three signers: Alice, Bob and Carol playing the roles of P_3, P_2 and P_1 respectively. For lack of space, we just illustrate the role of Alice. Alice starts the protocol by sending level 1 promises to Bob and Carol, and waits for level 2 promises from Bob and Carol. If Alice does not receive them, then Alice may contact T with a request to abort the exchange. If Alice does receive the promises, then she sends her level 3 promises to Bob and Carol, and waits for their level 3 promises in return. If Alice does not receive these promises then she contact T with a recovery request. Otherwise, she sends her signature on the preagreed text along with level 4 promises to Bob and Carol, and waits for their signatures. The protocol finishes for her when she receives the contract. Otherwise, she may launch the recovery subprotocol and contact T.

Abort protocol. T maintains two sets, S_m and F_m, that are used by T to make decisions when a signer contacts T. These sets are created when T is contacted for the first time for m and are initialized to be empty. The set S_m contains the indices of all signers that have contacted T and received an abort token from T in response. The intuitive meaning of the set F_m is not clearly stated in [12], but it contains some additional information that T uses in deciding when to overturn an abort decision that T has taken before.

The details of the abort protocol are given in table 2. Mainly, if T is contacted with a request to abort, then T checks its database. If this is the first request or if the protocol has not already been recovered, T sends back an abort token and updates the sets S_m and F_m. If the protocol has already been successfully recovered, T sends back a signed contract.

Recovery protocol. The details of the recovery protocol are given in table 3. For P_i to recover, it sends the message $S_{P_i}(\{PCS_{P_j}((m, k_j), P_i, T)\}_{j \in \{1,\ldots n\} \setminus \{i\}}, S_{P_i}((m, 1)))$, where

- if $j > i$, k_j is the maximum level of a promise received from P_j on m,

Table 1 GM multi-party contract-signing protocol—Main

Wait for all higher recursive levels to start

1. $P_j \rightarrow P_i$: $PCS_{P_j}((m,1),P_i,T)$ $(n \geq j > i)$

   If P_i does not receive 1-level promises from $P_n \ldots P_{i+1}$ in a timely manner, P_i simply quits.

Start recursive level i

2. $P_i \rightarrow P_j$: $PCS_{P_i}((m,1),P_j,T)$ $(i > j \geq 1)$

Wait for recursive level i-1 to finish

3. $P_j \rightarrow P_i$: $PCS_{P_j}((m,i-1),P_i,T)$ $(i > j \geq 1)$

   If P_i does not receive (i-1)-level promises from $P_{i-1} \ldots P_1$ in a timely manner, P_i aborts.

Send i-level promises to all lower-numbered signers

4. $P_i \rightarrow P_j$: $PCS_{P_i}((m,i),P_j,T)$ $(i > j \geq 1)$

Finish recursive level i when i-level promises are received

5. $P_j \rightarrow P_i$: $PCS_{P_j}((m,i),P_i,T)$ $(i > j \geq 1)$

   If P_i does not receive i-level promises from $P_{i-1} \ldots P_1$ in a timely manner, P_i recovers.

Complete all higher recursive levels

For $a = i+1$ to n, P_i does the following:

6.1. $P_i \rightarrow P_a$: $PCS_{P_i}((m,a-1),P_a,T)$

6.2. $P_j \rightarrow P_i$: $PCS_{P_i}((m,a),P_j,T)$ $(a \geq j > i)$

If P_i does not receive a-level promises from $P_a \ldots P_{i+1}$ in a timely manner, P_i recovers.

6.3. $P_i \rightarrow P_j$: $PCS_{P_i}((m,a),P_j,T)$ $(i > j \geq 1)$

6.4. $P_j \rightarrow P_i$: $PCS_{P_i}((m,a),P_j,T)$ $(i > j \geq 1)$

If P_i does not receive a-level promises from $P_{i-1} \ldots P_1$ in a timely manner, P_i recovers.

6.5. $P_i \rightarrow P_j$: $PCS_{P_i}((m,a),P_j,T)$ $(a \geq j > i)$

Wait for signatures and (n+1)-level promises from higher-numbered signers

7. $P_j \rightarrow P_i$: $PCS_{P_j}((m,n+1),P_i,T), S_{P_j}(m,1)$ $(n \geq j > i)$

   If P_i does not receive signatures and (n+1)-level promises from $P_n \ldots P_{i+1}$ in a timely manner, P_i recovers.

Send signatures and (n+1)-level promises to signers

8. $P_i \rightarrow P_j$: $PCS_{P_i}((m,n+1),P_j,T), S_{P_i}(m,1)$ $(j \neq i)$

Wait for signatures from lower-numbered signers

9. $P_j \rightarrow P_i$: $PCS_{P_j}((m,n+1),P_i,T), S_{P_j}(m,1)$ $(i > j \geq 1)$

   If P_i does not receive signatures and (n+1)-level promises from $P_{i-1} \ldots P_1$ in a timely manner, P_i recovers.

- if $j < i$, k_j is the maximum level of promises received from all signers $P_{j'}$, with $j' < i$, *i.e.*, the min-max of the level of promises from signers with lower index. (*E.g.*, if the maximum level of the promises received by P_4 from P_3 and P_2 was 6, and the maximum level received by P_4 from P_1 was 5, then it would send the 5-level promises for P_1, P_2 and P_3.)

If T is contacted with a request to recover, then T checks its database. If this is the first request for m or if the protocol has already been recovered, T replies with a signed contract which it obtains by converting the promises into conventional digital signatures. Otherwise, if the protocol has already been aborted, T must decide whether to maintain the abort or to overturn it. Overturning of the abort is necessary in order to maintain fairness. Indeed, consider the scenario in which a dishonest P_{n-1} contacts T with an abort request, receives an abort token and dishonestly continues the protocol. After the n-levels are completed, P_n sends its signature to others and waits for signatures from other signers. If P_{n-1} does not send back its signature, then P_n will be forced to contact T with a request to recover. Now, T must overturn its previous abort, otherwise P_n will not receive the signature of P_{n-1}. The decision whether to overturn is based on the contents of the sets S_m and F_m, as described in table 3.

2.2. BW multi-party contract-signing protocol

The structure of the BW protocol [4] is much simpler than the protocol discussed in section 2.1. The protocol consists of two subprotocols: main and recovery. Usually signers try to achieve the exchange by executing the main subprotocol. They contact T using the recovery subprotocol when they think something is amiss. For lack of space, we discuss the protocol briefly. The detailed description of the protocol is given in appendix A.

The main protocol is composed of $n+1$ rounds[1], and

Table 2 GM multi-party contract-signing protocol—Abort

1. $P_i \rightarrow$ T: $S_{P_i}(m, P_i, (P_1, \ldots, P_n), abort)$

   if not validated(m) then

   if $S_m = \emptyset$, T stores $S_T(S_{P_i}(m, P_i, (P_1, \ldots, P_n), abort))$; $S_m = S_m \cup \{i\}$;

   if i is larger than the maximum index in S_m, T clears F_m

   2. T $\rightarrow P_i$: $S_T(S_{P_j}(m, P_j, (P_1, \ldots, P_n), abort), S_T(m, S_m, abort))$

   else (validated(m)=true)

   3. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots,n\} \setminus \{i\}}$

   where k_j is the level of the promise from P_j that was converted to a universally-verifiable signature in the recovery protocol.

Table 3 GM multi-party contract-signing protocol—Recovery

1. $P_i \rightarrow$ T: $S_{P_i}(\{PCS_{P_j}((m, k_j), P_i, T)\}_{j \in \{1,\ldots n\} \setminus \{i\}}, S_{P_i}((m, 1)))$

   if $i \in S_m$, T ignores the message

   else if validated(m)

   2. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots n\} \setminus \{i\}}$

   where k_j is the level of the promise from P_j that was converted to a universally-verifiable signature.

   else if $S_m = \emptyset$

   validated(m):=true

   3. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots n\} \setminus \{i\}}$

   else (validated(m)=false $\wedge$ $S_m \neq \emptyset$)

   a. if $i \notin F_m$, then

      (i) if for any $\ell \in S_m$ there is a $j \in S_m$ such that $j > k_\ell$

      $S_m := S_m \cup \{i\}$

      let a be the maximum value in S_m

      if $a > i$ then $\forall j$, such that $k_j = a - 1 \cdot$ $F_m := F_m \cup \{j\}$

      else $F_m := \emptyset$

      4.1.1. T $\rightarrow P_i$: $S_T(S_{P_j}(m, P_j, (P_1, \ldots, P_n), abort), S_T(m, S_m, abort))$

      where $S_T(S_{P_j}(m, P_j, (P_1, \ldots, P_n), abort))$ corresponds to the stored abort token

      (ii) else

      validated(m):=true

      4.1.2. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots n\} \setminus \{i\}}$

   b. else ($i \in F_m$)

      let a be the maximum value in S_m

      (i) if $(\forall j$, such that $i < j \leq a \cdot$ $k_j < a) \wedge (\forall j < i \cdot$ $k_j \geq a)$

      validated(m):=true

      4.2.1. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots n\} \setminus \{i\}}$

      (ii) else

      $S_m := S_m \cup \{i\}$

      if $a > i$ then $\forall j, k_j = a - 1 \cdot$ $F_m := F_m \cup \{j\}$

      if $a = i$ then $F_m := \emptyset$

      4.2.2. T $\rightarrow P_i$: $S_T(S_{P_j}(m, P_j, (P_1, \ldots, P_n), abort), S_T(m, S_m, abort))$

      where $S_T(S_{P_j}(m, P_j, (P_1, \ldots, P_n), abort))$ corresponds to the stored abort token

is symmetric for each signer. In round i, a signer sends an i-level promise to other signers. The i-level promise is implemented using a universally-verifiable digital signature and includes all the promises received until round $i-1$. Note that these promises are based on universally verifiable signatures and that the protocol does not intend to provide abuse-freeness. The $n+1$ level promises from everybody is considered to be the signed contract. If any expected message is not received, a signer can launch the recovery protocol. Once the the signer launches recovery protocol, it is not allowed to continue the main subprotocol.

Once T is contacted by a signer, T responds with an abort token or a signed contract. T maintains a database of past requests and responses, and decides its future responses based upon this database. If the first request from any signer to T is in the first round, then T sends back an abort token. If the first request from any signer to T is in later rounds, then T sends back a signed contract. If T ever sends back a signed contract, it always sends back the signed contract. The abort decision may however be overturned. T overturns the abort decision if and only if T can conclude from the recovery request that all the signers that contacted T in the past dishonestly continued the main protocol after launching the recovery protocol. T concludes that a signer, say P_i, who contacted T in round r has dishonestly continued the protocol if some other signer presents T with a $r+1$ level promise from P_i. We believe that the design of this decision contributes to the robustness of the protocol.

3. Model

ATS, ATL and MOCHA. The desired properties of contract signing are easily described using games, and hence we chose a game-variant of Kripke structures, alternating transition systems (ATS) [1], to model the protocols. An ATS is composed of a set of players Σ, a set of states Q that represents all possible game configurations, a set $Q_0 \subseteq Q$ of initial states, a finite set of propositions Π, a labeling function $\pi : Q \to 2^{\Pi}$ that labels states with propositions, and a game transition function $\delta : Q \times \Sigma \to 2^{2^Q}$. For a player a and a state q, $\delta(q,a)$ is the set of choices that a can make in the state q. A choice is a set of possible next states. One step of the game at a state q is played as follows : each player $a \in \Sigma$ makes its choice and the next state of the game q' is the intersection (required to be a singleton) of the choices made by all the players of Σ, *i.e.*, $\{q'\} = \cap_{a \in \Sigma} \delta(q,a)$. A computation is an infinite sequence $\lambda = q_0 q_1 \ldots q_n \ldots$ of states obtained by starting the game in q_0, where $q_0 \in Q_0$.

In order to reason about ATS, we use alternating-time temporal logic (ATL) [1]. For a given set of players $A \subseteq \Sigma$, a set of computations Λ, and a state q, consider the following game between a protagonist and an antagonist starting in q. At each step, to determine the next state, the protagonist selects the choices controlled by the players in the set A, while the antagonist selects the remaining choices. If the resulting infinite computation belongs to the set Λ, then the protagonist wins. If the protagonist has a winning strategy, we say that the ATL formula $\langle\langle A \rangle\rangle \Lambda$ is satisfied in state q. Here, $\langle\langle A \rangle\rangle$ is a path quantifier, parameterized by the set A of players, which ranges over all computations that the players in A can force the game into, irrespective of how the players in $\Sigma \setminus A$ proceed. The set Λ is defined using temporal logic formulas. For those familiar with branching time temporal logics, the parameterized path quantifier $\langle\langle A \rangle\rangle$ can be seen as a generalization of the CTL path quantifiers: the existential path quantifier $\exists$ corresponds to $\langle\langle \Sigma \rangle\rangle$ and the universal path quantifier $\forall$ corresponds to $\langle\langle \emptyset \rangle\rangle$.

We now illustrate the expressive power of ATL allowing to model both cooperative and adversarial behavior amongst the players. Consider the set of players $\Sigma = \{a, b, c\}$ and the following formulas with their verbal reading:

- $\langle\langle a \rangle\rangle \Diamond p$, player a has a strategy against players b and c to eventually reach a state where proposition p is true;
- $\neg\langle\langle b, c \rangle\rangle \Box p$, the coalition of players b and c does not have a strategy against a to make p true forever;
- $\langle\langle a, b \rangle\rangle \bigcirc (p \wedge \neg\langle\langle c \rangle\rangle \Box p)$, a and b can cooperate so that the next state satisfies p and from there c does not have a strategy to impose p forever.

The details of ATS and ATL can be found in [1]. Instead of modeling protocols directly with ATS we use a more user-oriented notation: a guarded command language a la Dijkstra. The details about the syntax and semantics of this language (given in terms of ATS) can be found in [14]. Intuitively, each player $a \in \Sigma$ disposes of a set of guarded commands of the form $guard_\xi \to update_\xi$. A computation-step is defined as follows: each player $a \in \Sigma$ chooses one of its commands whose boolean guard evaluates to true, and the next state is obtained by taking the conjunction of the effects of each update part of the commands selected by the players. Given an ATS described in terms of guarded commands, the finite state tool MOCHA automates the model-checking of ATL formulae over the specified ATS.

Modeling protocols. Unlike the classical security protocols aiming at secrecy and authentication, optimistic contract-signing protocols usually consist of subprotocols that can be invoked at specified moments. Running a protocol at a time not foreseen by the designer, may have unexpected side-effects. This may be used by a signer to gain an advantage over other signers. We believe that such concurrency issues are a major source of problems. Therefore, and since the high number of messages would create a serious state explosion, we only analyze the *structure* of the protocols and

1 Actually, the protocol has $t+2$ rounds, where t is the maximum number of dishonest signers. In our analysis however we assume the worst possible case for an honest signer, namely that t is $n-1$.

concentrate only on single protocol instance. We now discuss our model in detail.

The protocol instance is modeled as an ATS and each protocol participant is modeled as a player in the ATS using the above introduced guarded command language. Besides, the branching aspect, another notable difference with more classical secrecy and authentication protocols, is that contract-signing protocols must be secure against malicious signer, rather than an external intruder. Therrefore, we have two process for each signer, one describing it's honest behavior and the other the dishonest one. Communication is modeled using shared variables. Each protocol message is modeled using a boolean variable, initialized to *false* and set to *true* when it is sent. Sending of a message is modeled using guarded commands, where the guard depends on previously sent out messages. When modeling the honest behavior of a participant, we ensure that a given message is sent out only when specified by the protocol. In contrary, the guards are relaxed in the malicious version of the signer so that each message can be sent out, as soon as possible, *i.e.*, as soon as all messages needed to compose the given message are received. We do not explicitly model any cryptographic primitives, but only the fact that protocol messages can be sent out of order. Hence, a dishonest signer can send messages out of order and continue the protocol, even if it is supposed to stop. We manually decide which messages must be known in order to send some other message. Moreover, the communication between any two signers is assumed to be on private channels and we do not model the possibility to spy other channel. Please note that in the original paper, communication between the signers is assumed to be in control of Dolev-Yao intruder. We have used a weaker attack model than in the original paper. This modeling is sound in the following sense: a scenario that violates a desirable property in this restricted communication model also violates the property in the general communication model. The trusted party is modeled to be always honest.

As an example, consider the short extract of the modeling of the three-party GM protocol depicted in figure 1. In the extract, the integer variable `Pr_i_j_L` models the promises that P_i has sent to P_j, and `Pr_i_j_L = k` means that P_i has sent out up to k-level promises to P_j. (For efficiency reasons, we use the logarithmic encoding of a ranged integer variable, rather than having one boolean variable for each level of promise). In the extract, the first rule of honest P_1 says that P_1 may quit the protocol, if it has not contacted the trusted party, and has neither received nor sent any promises. The corresponding modeling of dishonest behavior of P_1 states that P_1 may quit the protocol at any moment. The second rule of honest P_1 gives the exact condition when the first level promise has to be sent to P_2. The corresponding dishonest rule, merely requires that P_1 has not quit the protocol before sending the promise.

As we are unable to verify parametric systems with MOCHA, we simplify our task and verify the protocols only for a given number n of signers. Due to its complexity, encoding a protocol instance has revealed itself to be a time-consuming and challenging task. To avoid encoding each instance of the protocol using guarded commands, we have written a dedicated C++ program for each protocol which takes the number n of signers as a parameter and generates the protocol specification. Although our model is restricted with regard to several aspects, the model seems to be of interest as several unknown anomalies have been revealed.

Modeling properties. We express the desired security guarantees using ATL. For lack of space, we concentrate on modeling of fairness and discuss modeling of abuse-freeness in section 4. Consider an instance of the protocol with n signers, which we denote as $P_1, \ldots, P_n$. In the following, we assume that only one of the signers, say P_1 is honest, and the other dishonest signers are colluding to cheat the honest signer.

A protocol is fair for an honest P_1, if at the end of the protocol, either P_1 receives signed contracts from all the other signers or it is not possible for any other signer to obtain P_1's signed contract. One possible ATL formula for modeling this says that if any signer receives P_1's signed contract, then P_1 has a strategy to get the signed contracts from other signers:

$$\forall\Box((P_2.S_{P_1}(m) \vee \ldots \vee P_n.S_{P_1}(m)) \rightarrow \langle\langle P_1\rangle\rangle\Diamond(P_1.S_{P_2}(m) \wedge \ldots \wedge P_1.S_{P_n}(m))) \quad (1)$$

where $P_i.S_{P_j}(m)$ denotes that player P_i received P_j's signature on the contract text m.

It can of course be argued that P_1 having a strategy to receive the signed contracts is not a sound modeling of fairness: P_1 may have this strategy but if it is ignorant or if it mistakenly does not follow this strategy, then the protocol may end in an unfair state. Therefore one could require the following stronger property: in whatever way P_1 resolves the remaining choices specified by the protocol, P_1 receives all the signed contract.

$$\forall\Box((P_2.S_{P_1}(m) \vee \ldots \vee P_n.S_{P_1}(m)) \rightarrow \forall\Diamond(P_1.S_{P_2}(m) \wedge \ldots \wedge P_1.S_{P_n}(m))) \quad (2)$$

In the same vein, a third formulation only requires that there exists a path where P_1 receives the signed contracts.

$$\forall\Box((P_2.S_{P_1}(m) \vee \ldots \vee P_n.S_{P_1}(m)) \rightarrow \exists\Diamond(P_1.S_{P_2}(m) \wedge \ldots \wedge P_1.S_{P_n}(m))) \quad (3)$$

Formula (2) implies formula (1), which in turn implies formula (3). We concentrate on the last, weakest version of fairness. As we show that fairness is violated even in this weakest version, the other stronger versions are also violated. Now, we are ready to discuss our analysis.

Extract of honest modeling of P_1 for the three-party GM protocol:

```
[] ~P1_stop & ~P1_contacted_T & Pr_1_3_L=0 & Pr_1_2_L=0 & ~( Pr_3_1_L>0 & Pr_2_1_L>0 )
   -> P1_stop':=true
[] ~P1_stop & ~P1_contacted_T & Pr_1_3_L=0 & Pr_1_2_L=0 & Pr_2_1_L>0 & Pr_3_1_L>0
   -> Pr_1_2_L':=1
```

The corresponding actions of a dishonest modeling:

```
[] ~P1_stop -> P1_stop':=true
[] ~P1_stop & Pr_1_2_L<1 -> Pr_1_2_L':=1
```

Figure 1: Extract of the three-party GM protocol modeling

4. Analysis

We have verified the BW protocol with two and up to five signers, but the model-checker MOCHA did not find any flaw. Due to lack of space we do not go into the details of our analysis of the BW protocol. However, the design of the BW protocol is much simpler than the GM protocol and the decision to overturn an abort is based on the following argument: T overturns only if it can infer that no previous abort reply has been sent to a potentially honest principal. This seems to ensure the robustness of the protocol. Note that we only analyzed the structure of the protocol. Hence, our results only prove that the protocol is correct in the given model. Nevertheless, we believe that the protocol would also be correct in a more general model as all messages are signed and a unique contract identifier is used.

We now report in more detail on our the analysis of the GM contract-signing protocol. The protocol has several peculiarities. The most notable one is that the protocol changes with the number of signers, *e.g.*, the protocol specification of P_1 differs when the value of n changes. The number of protocol messages increases considerably with the number of signers. For instance, if we have $n = 3$, the main protocol has 20 messages and there are 14 different recovery requests. When $n = 4$, the corresponding numbers are 41 and 36. Moreover, the protocol is not symmetric for the signers: the protocol specification for P_i is different from that for P_j, for all $i \neq j$. For instance, when $n = 4$, P_1 can launch 18 different recovery requests and P_4 only 2.

As mentioned in section 3, we have written a dedicated C++ program that takes the number of signers, n, as a parameter and generates the protocol specification. Our analysis revealed problems with fairness, when n is 4. Although, we did not discover any fairness problems when $n = 3$, we did find an amusing problem with abuse-freeness. We did not discover any problems with timeliness in the protocol. All these anomalies are novel and the protocol was believed to be secure since it was first published. We discuss our results in detail. The sources codes of our analysis of both protocols are also available at the following website `http://www.ulb.ac.be/di/scsi/skremer/MPCS/`.

Fairness. We did not discover any problems with fairness when $n = 3$. The formulas representing fairness for P_1, P_2 and P_3, introduced in section 3, are validated by MOCHA. However, as we use a restricted model and consider single runs, we can only conclude that the protocol does not present any *structural* weakness for $n = 3$. Indeed, if we relax the assumption of the private channels, the anomaly presented by Shmatikov and Mitchell in [16] on the two-party GJM protocol can be adapted to the multi-party version. In this scenario, a malicious signer eavesdrops on the channel between the honest signer and T, and succeeds in compromising fairness. With our present modeling, we do not find such flaws, as this requires to eavesdrop channels and to decompose messages. The fix proposed by Shmatikov and Mitchell applies to the multi-party protocol too. However, we should emphasize that the authors of the GM protocol require the channels to T to be private and hence this scenario does not represent a valid attack on the protocol.

We discovered several scenarios that compromised fairness when $n = 4$. The first scenario was discovered by hand, when we found an error in the proof of correctness given in the original paper [12]. A detailed analysis using MOCHA detected seven other scenarios. An analysis of these revealed that the proof also did not cover a case. In each scenario, an honest signer is cheated by the coalition of three malicious signers. These scenarios follow the general outline:

1. A dishonest signer contacts the trusted party, T, at the beginning of the protocol, gets an abort token, and dishonestly continues participating in the main protocol.
2. A second dishonest signer tries to recover at some later point. It does not succeed, but manages to put the honest signer in the list F_m. It dishonestly continues the main protocol.
3. The honest signer is forced to recover, but is not successful in getting the abort decision overturned since it is in the list F_m.
4. The third dishonest signer contacts T and manages to overturn the decision. Hence, while the honest signer does not get any signed contract, the honest signer's contract is obtained.

For lack of space, we just describe one of these scenarios in detail. In this scenario, P_1, P_3 and P_4 collude to cheat P_2. The scenario proceeds as follows:

- At the beginning of the protocol, P_3 aborts the protocol and T updates $S_m = \{3\}$. However, unlike specified by the protocol, dishonest P_3 continues the main protocol execution.
- As soon as P_1 receives the second level promise from P_2, it asks T to recover by sending

$$S_{P_1}(\{PCS_{P_2}((m,2),P_1,T), PCS_{P_3}((m,1),P_1,T), PCS_{P_4}((m,1),P_1,T)\}, S_{P_1}((m,1))).$$

  T refuses this request, answers with an abort message and updates $S_m = \{1,3\}$ and $F_m = \{2\}$. As P_3 did before, P_1 also continues the protocol.
- The main protocol is executed normally until signer P_2 reaches point 6.2. (see table 1) of the protocol with $a = 4$. At that point P_2 has sent out the set of message

$$\{PCS_{P_2}((m,1),P_1,T), PCS_{P_2}((m,2),P_1,T), PCS_{P_2}((m,2),P_3,T), PCS_{P_2}((m,3),P_1,T), PCS_{P_2}((m,3),P_3,T), PCS_{P_2}((m,3),P_4,T)\}$$

  and has received the set of messages

$$\{PCS_{P_4}((m,1),P_2,T), PCS_{P_3}((m,1),P_2,T), PCS_{P_1}((m,1),P_2,T), PCS_{P_1}((m,2),P_2,T), PCS_{P_3}((m,3),P_2,T), PCS_{P_1}((m,3),P_2,T)\}.$$

  P_2 is at position 6.2. with $a = 4$ and is waiting for 4-level promises from P_3 and P_4. P_3 and P_4 do not reply and P_2 is forced to send the following recovery request to T.

$$S_{P_2}(\{PCS_{P_2}((m,3),P_2,T), PCS_{P_3}((m,3),P_2,T), PCS_{P_4}((m,1),P_2,T)\}, S_{P_2}((m,1))).$$

  P_2 is in F_m, the tests in the protocol description (see table 3) indicate that T refuses the request, updates $S_m = \{1,2,3\}$ and replies with an abort message. F_m remains unchanged.
- P_4 launches a resolve request, sending

$$S_{P_4}(\{PCS_{P_1}((m,3),P_4,T), PCS_{P_2}((m,3),P_4,T), PCS_{P_3}((m,3),P_4,T)\}, S_{P_4}((m,1))).$$

  This request overturns the previous aborts and violates fairness as T sends the signed contract back to P_4.

We also discovered that the protocol is unfair for signers P_1, P_2 and P_3 when $n = 4$. We did not find any scenario that violates fairness for P_4. This is probably because the tests indicate that P_4 cannot be added to F_m, when $n = 4$.

Correcting the Garay-MacKenzie Protocol. In order to restore fairness in the Garay-MacKenzie protocol, we had to do major revisions in the recovery protocol. We were unsuccessful to restore fairness with minor changes, and we believe that this is because the meaning of the list F_m is not clear in the protocol. The central idea behind the revision is that T, when presented with a recovery request, overturns its abort decision if and only if T can infer dishonesty on the part of each of the signer that contacted T in the past. This is also the main idea behind the recovery protocol in [4].

The main protocol remains the same. Major changes are in the recovery protocol. The recovery messages are designed so that T can infer the promises that an *honest* signer would have sent when it launched the recovery protocol (note that a signer may have dishonestly sent other promises). For P_i to recover, it sends the message

$$S_{P_i}(\{PCS_{P_j}((m,k_j),P_i,T)\}_{j\in\{1,\ldots n\}\setminus\{i\}}, S_{P_i}((m,1)))$$

where k_j is computed as following:

1. If P_i runs the resolve protocol in step 5 of the main protocol (see table 1), then $k_j = 1$ for $j > i$ and $k_j = i-1$ for $j < i$.
2. In step 6.2 of the main protocol, $k_j = a - 1$ for $1 < j \leq a-1, j \neq i$ and $k_j = 1$ for $j > a-1$.
3. In step 6.4 of the main protocol, $k_j = a - 1$ for $j < i$, $k_j = a$ for $i < j \leq a$ and $k_j = 1$ for $j > a$.
4. In step 7 of the main protocol, $k_j = n$ for all j.
5. In step 9 of the main protocol, $k_j = n$ for all $j < i$ and $k_j = n + 1$ for all $j > i$.

k_j may alternately be computed as:

- If $j < i$, k_j is the maximum level of promises received from all signers $P_{j'}$, with $j' < i$, *i.e.* the min-max of the promises from signers with lower index. (For example, if the maximum level of the promises received by P_4 from P_3 and P_2 was 6, and the maximum level received by P_4 from P_1 was 5, then it would send the 5-level promises for P_1, P_2 and P_3.)
- Let l be the maximum value l' such that P_i has l' level promises from P_j for all $i \leq j \leq l'$. If no such l' exists then let l be 0. If $l = 0$, then let $k_j = 1$ for all $j > i$. If $l \neq 0$, then let $k_j = l$ for all $i \leq j \leq l$ and $k_j = 1$ for all $j > l$. (E.g., if P_2 has received level 1 promise from P_6, level 5 and 1 promises from P_5, level 5, 4 and 1 promises from P_4, and level 4, 3 and 1 promises from P_3 then $k_6 = 1, k_5 = 1, k_4 = 4, k_3 = 4$.)

T maintains the set S_m of indices of signers that contacted T in the past and received an abort token. For each signer P_i in the set S_m, T also maintains two integer variables $h_i(m)$ and $l_i(m)$. Intuitively, h_i is the highest level promise an honest P_i could have sent to any higher indexed signer before it contacted T. l_i is the highest level promise an honest P_i could have sent to a lower indexed signer before it contacted T. The protocols for T works as follows:

- If T ever replies with a signed contract for m, then T responds with the contract for any further request.
- If the first request to T is a resolve request, then T sends back a signed contract.
- If the first request is an abort request, then T aborts the contract. T may overturn this decision in the future if it can deduce that all the signers in S_m have behaved dishonestly. T deduces that a signer P_i in S_m is dishonest when contacted by P_j if
  1. $j > i$ and P_j presents to T a k-level promise from P_i such that $k > h_i(m)$, or
  2. $j < i$ and P_j presents to T a k-level promise from P_i P_i such that $k > l_i(m)$.

The detailed abort recovery subprotocols are given in table 4 and 5 respectively. We analyzed the revised protocol for both 3 and 4 signers and MOCHA did not detect any errors in the revised protocol. Please note that this should not be construed as proof of correctness since we are using a restricted communication model and are modeling a single run. Nevertheless, we believe that the revised protocol would be fair in a more general setting, and for an arbitrary number of signers.

Abuse-freeness. We describe the anomaly that we discovered for $n = 3$ signers in the GM protocol. It exploits the fact that when T replies with an abort decision, it also signs the list S_m of the signers who have received an abort from T. Recall that an optimistic signer [8] is one that prefers to wait for "some time" before contacting the trusted party. Following [8], we say that a protocol is abuse-free for a signer P_i if the protocol does not provide *provable advantage* to the remaining signers. A coalition of signers is said to have provable advantage against P_i at a point in the protocol if (i) they have a strategy to abort the contract against an optimistic P_i, (ii) they have a strategy to get optimistic P_i's contract, and (iii) they can prove to an outside challenger, Charlie, that P_i is participating in the protocol.

Now consider the protocol instance with three signers P_1, P_2 and P_3. Assume that P_3 is optimistic and P_1 and P_2 are colluding to cheat P_3. P_3 starts the protocol by sending its level 1 promises to P_1 and P_2, and waits for level 2 promises from them. P_2 on receiving this sends its level 1 promise to P_1, and then sends an abort request to T which aborts the protocol. Now, P_1 has received level 1 promises from P_2 and P_3. Using these first level promises, P_1 sends a recovery request to T. Note that, in the protocol, P_1 is never allowed to abort and T would not accept an abort request from P_1. P_1's recovery request is refused and T sends

$$S_T(S_{P_2}(m, P_2, (P_1, P_2, P_3), abort))$$

and

$$S_T(m, S_m = \{1, 2\}, abort)$$

At this point, we make the following observations:

- the abort reply contains the set $S_m = \{1, 2\}$ and is different from the one P_2 received,
- if P_1 receives an abort reply from T, it is always the answer to a recovery request,
- a recovery request always includes a promise from each signer which is verified by T.

From these remarks, we can conclude that if P_1 shows the abort reply to Charlie, Charlie will be convinced that P_3 has started the protocol even though Charlie is unable to verify the PCS from P_3. In other words, we can say that T has verified the PCS for Charlie. At this point P_1 and P_2 can force the exchange to abort by simply quitting the protocol: P_3 has no promises from P_1 and P_2. P_1 and P_2 can also force a successful completion of the contract exchange by simply (dishonestly) engaging P_3 in the main protocol. Hence the protocol is not abuse-free for P_3.

This vulnerability can be addressed by excluding the set S_m from the abort reply. In this case, the abort messages from P_3 and P_2 are exactly similar and can be obtained by P_2 without P_3's participation. Hence, an abort reply does not prove P_3's participation in the protocol. This rather amusing scenario illustrates that sometimes additional information may be harmful. While explicitness is often considered a good engineering practice (and we do not attempt to criticize such thumb rules), care should be taken when applying these principles. In personal communication with the authors of the protocol, they propose a different fix in letting P_1 abort the protocol rather than just quitting.

Abuse-freeness for P_3 is naturally expressed in ATL as follows:

$$\begin{array}{l}\neg\exists\Diamond(T.send(abort)\ to\ P_1 \wedge \\ \quad \langle\!\langle P_1, P_2\rangle\!\rangle \Box(\neg P_3.S_{P_1}(m) \vee \neg P_3.S_{P_2}(m)) \wedge \\ \quad \langle\!\langle P_1, P_2\rangle\!\rangle \Box(P_3.stop \rightarrow (P_1.S_{P_3}(m) \wedge P_2.S_{P_3}(m))) \\ )\end{array}$$

The boolean variable $T.send(abort)\ to\ P_1$ is set to $true$ when P_1 receives the abort token. As discussed before, this serves as a proof of P_3's participation. The variables $P_i.S_{P_j}(m)$ reflect that player i has received player j's signature on the contract. More precisely, the formula requires that it is not possible to reach a point where

1. P_1 and P_2 can prove to Charlie that the protocol was started by P_3 ($T.send(abort)\ to\ P_1$ is true),
2. P_1 and P_2 have a strategy to choose an unsuccessful outcome, *i.e.*, P_3 cannot get some signer's contract ($\langle\!\langle P_1, P_2\rangle\!\rangle \Box(\neg P_3.S_{P_1}(m) \vee \neg P_3.S_{P_2}(m))$ is true), and
3. P_1 and P_2 have a strategy to choose a successful outcome, *i.e.*, when honest P_3 stops, they have obtained P_3's contract ($\langle\!\langle P_1, P_2\rangle\!\rangle \Box(P_3.stop \rightarrow (P_1.S_{P_3}(m) \wedge P_2.S_{P_3}(m))$ is true).

Table 4 Revised GM multi-party contract-signing protocol—Abort

1. $P_i \rightarrow$ T: $S_{P_i}(m, P_i, (P_1, \ldots, P_n), abort)$

   if not validated(m) then

   $S_m = S_m \cup \{i\}$;

   if $S_m = \emptyset$, T stores $S_T(S_{P_i}(m, P_i, (P_1, \ldots, P_n), abort))$;

   2. T $\rightarrow P_i$: $S_T(S_{P_j}(m, P_j, (P_1, \ldots, P_n), abort), S_T(m, S_m, abort))$

   else (validated(m)=true)

   3. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots,n\} \setminus \{i\}}$

   where k_j is the level of the promise from P_i that was converted to a universally-verifiable signature during the recovery protocol.

Table 5 Revised GM multi-party contract-signing protocol—Recovery

1. $P_i \rightarrow$ T: $S_{P_i}(\{PCS_{P_j}((m, k_j), P_i, T)\}_{j \in \{1,\ldots n\} \setminus \{i\}}, S_{P_i}((m, 1)))$

   if $i \in S_m$, T ignores the message

   else if validated(m)

   2. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots n\} \setminus \{i\}}$

   where k_j is the level of the promise from P_j that was converted to a universally-verifiable signature.

   else if $S_m = \emptyset$

   validated(m):=true

   3. T $\rightarrow P_i$: $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots n\} \setminus \{i\}}$

   else (validated(m)=false $\wedge$ $S_m \neq \emptyset$)

   1. If there is some $p < i$ in S_m such that $k_p \leq h_p(m)$, or if there is some $p > i$ in S_m such that $k_p \leq l_p(m)$, then T sends back the stored abort $S_T(S_{P_j}(m, P_j, (P_1, \ldots, P_n), abort))$ to P_i. T adds i to S_m, and computes $h_i(m)$ and $l_i(m)$ as follows

      $(h_i(m), l_i(m)) = (k_{i+1}, 0)$, if $i = 1$ (intuitively, P_1 has contacted T in either step 6.2 of the main protocol with $a = k_{i+1} + 1$ or in step 7 of the main protocol),

      $= (0, i)$, if $1 < i$ and $k_{i-1} = i - 1$ (intuitively, P_i has contacted T in step 5 of the main protocol),

      $= (k_{i-1}, k_{i-1})$, if $1 < i < n, i \leq k_{i-1} \leq n$ and $k_{i+1} \leq k_{i-1}$ (intuitively, P_i has contacted T in step 6.2 of the main protocol with $a = k_{i-1} + 1$),

      $= (k_{i-1}, k_{i-1} + 1)$, if $1 < i < n, i \leq k_{i-1} < n$ and $k_{i+1} > k_{i-1}$ (intuitively, P_i has contacted T in step 6.4 of the main protocol with $a = k_{i-1} + 1$),

      $= (n, n)$, if $1 < i < n$ and $k_{i-1} = k_{i+1} = n$. (intuitively, P_i has contacted T in step 7 of the main protocol).

      $= (n + 1, n + 1)$, if $1 < i < n, k_{i-1} = n$ and $k_{i+1} = n + 1$. (intuitively, P_i has contacted T in step 9 of the protocol).

      $= (0, n + 1)$, if $i = n$ and $k_{i-1} = n$. (intuitively, P_n has contacted T in step 9 of the main protocol).

   2. Otherwise, T sends $\{S_{P_j}((m, k_j))\}_{j \in \{1,\ldots n\} \setminus \{i\}}$ to P_i, stores all the signatures, and sets validated(m) to true.

The requirement of P_3 stopping in condition 3 is to prevent it from idling forever. Even though as discussed before, the protocol is not abuse-free if P_3 is optimistic, the above formula is validated if P_3 is honest. An honest P_3 may contact T non-deterministically as permitted by the protocol. Indeed, in the scenario discussed above, an honest P_3 could prevent P_1 and P_2 from getting P_3's signature if it contacts T. Therefore, in order to capture the scenario described above, we needed to model optimistic signers.

Following [8], we implement an optimistic signer by adding *signals* that the signer uses to decide when to quit waiting for messages from other signers and contact T. P_3 uses 3 signals for this: one to decide when to ask T to abort and 2 to decide when to contact T for the two recovery protocols that P_3 can launch. These signals are controlled by a new player, $P3TimeOuts$, that is added to the model.

The decision to abort is modeled by 2 boolean variables: $setTimeOutAbort$ and $expiredTimeOutAbort$. While P_3 changes the value of $setTimeOutAbort$, $expiredTimeOutAbort$ is changed by $P3TimeOuts$. When P_3 sends level 1 promise to P_1 and P_2, it sets the value of $setTimeOutAbort$ to $true$, and then waits for level 2 promises from them. $P3TimeOuts$ may set $expiredTimeOutAbort$ to $true$ once $setTimeOutAbort$ is set to $true$ by P_3. If the promises arrive before $expiredTimeOutAbort$ is $true$, then P_3 continues with the main protocol, otherwise P_3 may contact T with an abort request. The decision to send recovery requests are modeled similarly.

Following [8], abuse-freeness is modeled by having a coalition of $P3TimeOuts$, P_1 and P_2. This coalition can choose a sufficiently "long time" to keep P_3 from contact-

ing T, while allowing P_1 and P_2 to schedule its messages in order to get the desired result. Abuse-freeness can then be expressed as

$$\begin{array}{l}\neg\exists\Diamond(T.\,send(abort)\ to\ P_1\wedge\\ \quad\langle\langle P_1, P_2, P3TimeOuts\rangle\rangle\Box\\ \qquad\quad(\neg P_3.S_{P_1}(m)\vee\neg P_3.S_{P_2}(m))\wedge\\ \quad\langle\langle P_1, P_2, P3TimeOuts\rangle\rangle\Box\\ \qquad\quad(P_3.stop\rightarrow(P_1.S_{P_3}(m)\wedge P_2.S_{P_3}(m)))\\ )\end{array}$$

Please note that even an optimistic P_3 should eventually be allowed to contact T, otherwise P_3 may be stuck forever. Hence, $P3TimeOuts$ must eventually set $expiredTimeOutAbort$ and other signals to $true$. Ideally, this should be set non-deterministically. However, ensuring that a variable changes its value in MOCHA slows down verification considerably. In order to make the verification feasible, we put a maximum limit, $tick$, on the number of computation steps after which the value must change, and vary this limit manually. Please note that the signals may change before this limit is reached. This modeling is sound in the sense that if the formula is violated for some value of $tick$ then abuse-freeness must be violated: P_3 just needs to wait for sufficiently "long time" to allow P_1 and P_2 to schedule its messages. Indeed, the above property is violated when $tick$ is set to 3 giving us the attack on abuse-freeness. As expected, if we drop the player $P3TimeOuts$ in the formula, then the property is not violated: P_1 and P_2 are not able to schedule their messages ahead of P_3.

We have also shown that a stronger version of abuse-freeness, introduced in [15], is violated, even in a non-optimistic setting.

5. Conclusions and Future Work

We have studied two multi-party contract-signing protocols [12, 4] using a finite-state tool, MOCHA, that allows specification of properties in a branching-time temporal logic with game semantics. In order to make this analysis feasible, we model single runs and assume a restricted communication model. Our analysis did not find any errors in the BW protocol [4]. We did encounter problems with fairness in the case of four signers in the GM protocol [12]. It appears that fairness cannot be restored without completely rewriting the subprotocols. The revised subprotocols are inspired by the BW protocol. We also discovered a rather amusing problem with abuse-freeness in the GM protocol with three signers that occurs because abort messages from the trusted party reveal who have contacted it in the past. This problem is easily addressed by ensuring that the trusted party does not send this extra information. We had to implement optimistic signers to demonstrate this problem using MOCHA.

We modeled single runs, and used a restricted communication and cryptographic model for our analysis. Previous work on two-party contract signing protocols has shown that they are prone to error in a more general setting. For example, in [6, 13], the authors exhibit problems with fairness when multiple sessions are involved, and in [3], the authors exhibit errors when black-box cryptography is replaced by provably secure cryptographic signature schemes. We plan to verify the protocols without fixing the number of signers. One major challenge in such a parametric verification is that the protocol descriptions change fundamentally with the number of signers in that the protocol for n signers is not merely putting n identical processes in parallel. We hope to prove the correctness of these protocols in a more general setting which accounts for cryptography, multiple concurrent sessions, and relaxes the communication model. We plan to use, at least partially, abstraction techniques such as proposed by Das and Dill [9] to achieve this.

References

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *38th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1997.

[2] N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In *4th ACM Conference on Computer and Communications Security*, Zurich, Switzerland, Apr. 1997. ACM Press.

[3] M. Backes, B. Pfitzmann, and M. Waidner. Reactively secure signature schemes. In *6th Information Security Conference (ISC)*, volume 2851 of *Lecture Notes in Computer Science*, pages 84–95, 2003.

[4] B. Baum-Waidner and M. Waidner. Round-optimal and abuse free optimistic multi-party contract signing. In *Automata, Languages and Programming — ICALP 2000*, volume 1853 of *Lecture Notes in Computer Science*, pages 524–535, Geneva, Switzerland, July 2000. Springer-Verlag.

[5] H. Burk and A. Pfitzmann. Value exchange systems enabling security and unobservability. In *Computers and Security, 9(8):715–721*, 1990.

[6] R. Chadha, M. Kanovich, and A. Scedrov. Inductive methods and contract-signing protocols. In *8th ACM Conference on Computer and Communications Security*, Philadelphia, PA, USA, Nov. 2001. ACM Press.

[7] R. Chadha, S. Kremer, and A. Scedrov. Formal analysis of multi-party contract signing. In *Workshop on Issues in the Theory of Security — WITS'04*, Barcelona, Spain, Apr. 2004. Accepted for publication.

[8] R. Chadha, J. C. Mitchell, A. Scedrov, and V. Shmatikov. Contract signing, optimism, and advantage. In *CONCUR 2003 — Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[9] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Sixteenth Annual IEEE Symposium on*

Logic in Computer Science (LICS 01), pages 51–60. IEEE Computer Society Press, June 2001.

[10] S. Even and Y. Yacobi. Relations among public key signature systems. Technical Report 175, Technion, Haifa, Israel, Mar. 1980.

[11] J. A. Garay, M. Jakobsson, and P. D. MacKenzie. Abuse-free optimistic contract signing. In *Advances in Cryptology—Crypto 1999*, volume 1666 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[12] J. A. Garay and P. D. MacKenzie. Abuse-free multi-party contract signing. In *International Symposium on Distributed Computing*, volume 1693 of *Lecture Notes in Computer Science*, Bratislava, Slavak Republic, Sept. 1999. Springer-Verlag.

[13] S. Gürgens and C. Rudolph. Security analysis of (un-) fair non-repudiation protocols. In A. E. Abdallah, P. Ryan, and S. A. Schneider, editors, *Formal Aspects of Security*, volume 2629 of *Lecture Notes in Computer Science*, pages 97–114, London, UK, 2003. Springer-Verlag.

[14] T. A. Henzinger, R. Manjumdar, F. Y. Mang, and J.-F. Raskin. Abstract interpretation of game properties. In *SAS 2000: Intertional Symposium on Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 220–239, Santa Barbara, USA, June 2000. Springer-Verlag.

[15] S. Kremer and J.-F. Raskin. Game analysis of abuse-free contract signing. In *15th IEEE Computer Security Foundations Workshop*, Cape Breton, Canada, June 2002. IEEE Computer Society Press.

[16] V. Shmatikov and J. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science, special issue on Theoretical Foundations of Security Analysis and Design*, 283(2):419–450, 2002.

A. Baum-Waidner multi-party contract-signing protocol

The protocol allows n ($n \geq 2$) participants or *signers*, say $P_1, \ldots, P_n$, to exchange signatures with the help of a trusted party T on a preagreed contract text m. In our description, we suppose $n-1$ potentially dishonest signers. The original protocol is actually parameterized with respect to a threshold t, the maximum number of possibly dishonest signers. In our analysis however we assume the worst possible scenario for an honest signer, namely that all the other signers are dishonest (*i.e.*, t is $n-1$).

The protocol consists of two subprotocols: main and recovery. Usually signers try to achieve the exchange by executing the main subprotocol. They contact T using the recovery subprotocol when they think something is amiss.

Main protocol. The main protocol for each signer P_i is given in table 6. The protocol is symmetric for each signer and is composed of $n+1$ rounds[2]. In each round, a signer sends a promise to other signers. The level of the promise is increased in each round, and considered as a signed contract once the round number equals $n+1$. The promise is implemented using a universally-verifiable digital signature includes the history of all previously received promises, through the vectors M and X, as defined in table 6. If any expected message is not received, P_i can decide to launch a recovery protocol.

Recovery Protocol The details of the recovery protocol are given in table 7. If the recovery request is launched in the first round, *i.e.*, P_i did not receive a message from all the signers, the recovery request consists of the first level promise of P_i. Otherwise, if $r > 1$, the recovery request contains, via the vector $X_{r-1,i}$ (see the main subprotocol in table 6), the set of received messages until round $r-1$, including the $r-1$ promises from all the other signers.

T maintains a variable, recovered, that indicates whether the given contract has been successfully recovered or not. It also maintains a set con, containing the indices of the signers that contacted T for m, and a set abort_set containing the indices of the signers for whom T aborted the protocol.

T ignores a recovery request from a signer if the signer has contacted T in the past. Otherwise, it checks whether the contract has already been successfully recovered or not. A successful recovery is always maintained. Otherwise, there are two cases. If the recovery request is sent in the first round, T must abort the protocol, as the request does not contain a proof that all signers actually started the protocol. If the recovery request is sent during any later round, say r, then T checks if all the requests that were aborted previously occurred at least two rounds before. If so, T can be sure that all the were sent to signers who dishonestly continued the main protocol. This is because the recovery request contains $r-1$ promises from all these signers, which they are not allowed to be send if they contacted T in round $r-2$ or before. Hence, the previous abort decision is overturned. Otherwise, T replies with an abort token.

2 In [4], the protocol has $t+2$ rounds.

Table 6 Baum-Waidner multi-party contract signing protocol—Main

$r := 1$

1. $P_i \rightarrow P_j$: $m_{1,i} = S_{P_i}(c, 1, \text{prev\_round\_ok})(j \neq i)$
2. $P_j \rightarrow P_i$: $m_{1,j} = S_{P_j}(c, 1, \text{prev\_round\_ok})$ $(j \neq i)$

if P_i times out then recovery(1)

P_i computes vectors $M_{1,i} := (m_{1,1}, \ldots, m_{1,n})$ and $X_{1,i} := M_{1,i}$

for $r := 2$ to $n + 1$ do

3. $P_i \rightarrow P_j$: $m_{r,i} = S_{P_i}(M_{r-1,i}, r, \text{vec\_ok}), S_{P_i}(c, r, \text{prev\_round\_ok})(i \neq j)$
4. $P_j \rightarrow P_i$: $m_{r,j} = S_{P_j}(M_{r-1,j}, r, \text{vec\_ok}), S_{P_j}(c, r, \text{prev\_round\_ok})$ $(j \neq i)$

if P_i times out then recovery(r)

P_i computes vectors

$$M_{r,i} := (S_{P_1}(c, r, \text{prev\_round\_ok}), \ldots, S_{P_n}(c, r, \text{prev\_round\_ok}))$$

and

$$X_{1,i} := (S_{P_1}(M_{r-1,1}, r, \text{vec\_ok}), \ldots, S_{P_n}(M_{r-1,n}, r, \text{vec\_ok}))$$

Table 7 Baum-Waidner multi-party contract signing protocol—Recovery

1. $P_i \rightarrow$ T: $\text{resolve}_{r,i}$

where

$$\text{resolve}_{r,i} = \begin{cases} (1, i, S_{P_i}(m_{1,i}, \text{resolve})) & \text{if } r = 1 \\ (r, i, S_{P_i}(X_{r-1,i}, \text{resolve})) & \text{otherwise} \end{cases}$$

if $i \in$ con then stop

else if recovered

con:=con$\cup\{i\}$

2. T $\rightarrow P_i$: $\text{signed}_{r,i}$

where $\text{signed}_{r,i} = \text{first\_signed}$

else ($\neg$recovered)

a. if $r = 1$

abort_set:=abort_set $\cup\{(r, i)\}$

con:=con$\cup\{i\}$

3. T $\rightarrow P_i$: $\text{aborted}_{r,i}$

where $\text{aborted}_{r,i} = S_T(c, r, i, \text{aborted})$

b. else ($r > 1$)

(i) if $\forall(s, k) \in \text{abort\_set} \cdot s < r - 1$

if con=$\emptyset$ then T sets first_signed:=$(\text{resolve}_{r,i}, S_T(c, r, i, \text{recovered}))$

recovered:=true

con:=con$\cup\{i\}$

4. T $\rightarrow P_i$: $\text{signed}_{r,i}$

where $\text{signed}_{r,i} = \text{first\_signed}$

(ii) else

abort_set:=abort_set $\cup\{(r, i)\}$

con:=con$\cup\{i\}$

5. T $\rightarrow P_i$: $\text{aborted}_{r,i}$

where $\text{aborted}_{r,i} = S_T(c, r, i, \text{aborted})$

Assembling Components with Behavioural Contracts
Assemblage de Composants selon des Contrats Comportementaux

Cyril Carrez[1,2] — Alessandro Fantechi[3,4] — Elie Najm[2]

[1] *Department of Telematics, Norwegian University of Science and Technology*
O.S. Bragstads plass 2B, N-7491 Trondheim, Norway
cyril.carrez@item.ntnu.no

[2] *Ecole Nationale Supérieure des Télécommunications, Département INFRES*
46 rue Barrault, F-75013 Paris, France
elie.najm@enst.fr

[3] *Universitá di Firenze, Dipartimento di Sistemi e Informatica*
Via S. Marta 3, I-50139 Firenze - Italy
fantechi@dsi.unifi.it

[4] *ISTI - CNR; Via G. Moruzzi 1, I-56124 Pisa - Italy*

ABSTRACT. *Component based design is a new paradigm to build distributed systems and applications. The problem of compositional verification of such systems is however still open. We investigate methods and concepts for the provision of "sound" assemblies. We define a behavioural interface type language endowed with a (decidable) set of interface compatibilty and subtyping rules. We define an abstract, dynamic, multi-threaded, component model, encompassing both client/server and peer to peer communication patterns. Based on the notion of compliance of components to their interfaces, we define the concepts of "contract" and "contract satisfaction". This leads to sound assemblies of components, which possess intersting properties, such as "external deadlock freeness" and "message consumption".*

RÉSUMÉ. *La conception basée composants est une nouvelle méthode de construction d'applications et de systèmes distribués. La vérification compositionelle de ces systèmes reste cependant un problème. Nous étudions des méthodes et des concepts pour la construction d'assemblages "sains". Nous définissons un langage de type d'interfaces comportementales, doté d'un ensemble de règles (décidables) de compatibilité d'interface et de sous-typage. Nous définissons un modèle de composant abstrait, dynamique et multi-tâche, qui englobe les modèles client/serveur et point-à-point. Basé sur la notion de conformité du composant par rapport à son interface, nous définissons les concepts de "contrat" et "respect de contrat". Cela mène aux assemblages sains de composants qui possèdent des propriétés intéressantes, comme "l'absence d'interblocage externe" et "la consommation des messages".*

KEYWORDS: *Behavioural typing, verification, composition, components, peer-to-peer, client/server*

MOTS-CLÉS : *Typage comportemental, vérification, composition, composants,* peer-to-peer, *client/serveur*

1. Introduction

Behavioural type systems have been defined in recent years with the aim to be able to check the compatibility of communicating concurrent objects, not only regarding data exchanged, but also regarding the matching of their respective behaviour [Nie95, KPT99, NNS99]. This check finds a natural application in the verification of compatibility of components, as the recent advances in Software Engineering are towards *component-based design*: a software system is developed as a construction based on the use of components connected together either by custom-made glue code, or by resorting to a standard platform supporting composition and communication, such as CORBA or .NET. The compatibility of a component with its environment has to be guaranteed before it is deployed.

Formal verification techniques can therefore play a strategic role in the development of high quality software. In the spirit of the so-called *lightweight formal methods*, the software engineer who connects components is not bothered by a formal description of the software artifact he is building, but gets a guarantee about the absence of mismatches between components; this guarantee is provided by the underlying formally verified components and by the formal verification algorithms that check type compatibility. An even more demanding example is mobile code, where one needs the guarantee that a migrating component does not undermine the correctness of the components that it reaches. This check has to be performed at run-time, at the reception of the migrating component, and hence has to be performed very efficiently. Typing of mobile agents has already been addressed for example in [HR02], but we aim at a more abstract behaviour of the component which is sufficient to efficiently prove that desired properties of the global configuration of components are not endangered by the composition.

In this work we define a framework in which a component can exhibit several *interfaces* through which it communicates with other components. Each interface is associated a type, which is an abstraction of the behaviour of the component. Our type language (for interfaces) introduces modalities on the sequences of actions to be performed by interfaces. Using **must** and **may** prefixes, it allows the distinction between *required* messages and *possible* ones. The complexity of the interface typing language is kept deliberately low, in order to facilitate compatibility verification among interfaces. We do not give a specific language for components, but we rather give an abstract definition, which wants to be general enough to accomodate different languages: indeed, components are abstracted as a set of ports, by which they communicate, together with a set of internal threads of execution, of which we observe only the effects on the ports. Under given constraints on the use of ports inside components, it is shown that a configuration made up of communicating components satisfies well-typedness and liveness properties, if the components honour the contracts given them by their interfaces, and the communicating interfaces are compatible. This work extends our previous one [CFN03b, CFN03a] with subtyping and a dual relation, and how they relate to the compatibility relation; we also put our work into practice with a more complex example.

Our work is partly inspired by that of De Alfaro and Henzinger [dAH01], who associate interface automata to components and define compatibility rules between interfaces. Our approach, which belongs instead to the streamline of process algebraic type systems, brings in the picture also the compliance between components and interfaces: the interface is thought as a *contract* with the environment, that the component should honour. We also aim at limit-

ing as much as possible the complexity of the interface compatibility check, which can even be needed to be performed at run-time. The work on Modal Transition Systems by Larsen, Steffen and Weise [LSW95] has inspired our definition of modalities and the way interface compatibility is checked. The guarantee of the satisfaction of well-typedness and liveness properties has been dealt by Najm, Nimour and Stefani [NNS99], and we have inherited their approach in showing how the satisfaction of compatibility rules guarantees more general properties.

This paper is structured as follows: in Section 2 we show our interface language and the related compatibility and subtyping rules, whereas Section 3 deals with the reference component model. Interface language and component model are put together in the concept of *component honouring a contract* (Sect. 4), which leads to interesting properties that can be guaranteed by sound assemblies of components (Sect. 5). Our work is illustrated with the classic bank account example in Section 6.

2. Interface Types

In this section we describe the language used to define the interfaces. A typed component is a component whereby every initial port or created port has an associated type, and every reference to a port has a declared type. We adopt a behavioral type language [Nie95, KPT99, NNS99]. In this setting, the type of a port prescribes its possible states, and for each state, the actions that are allowed and/or required through that port, and its state after the performance of an action. The BNF table below defines the syntax of types. Among the salient features of this type language is the use of **may** and **must** modalities. We also present a subtype relation and a compatibility relation.

2.1. *Syntax of the Interface Language*

The interface language has the following syntax:

$$
\begin{array}{rcl}
\mathit{type} & ::= & \mathit{server\_name}\ \textbf{=}\ \mathit{mod}\ \mathit{receive\_server} \\
 & | & \mathit{peer\_name}\ \ \textbf{=}\ (\mathit{mod}\ \mathit{send} \mid \mathit{mod}\ \mathit{receive}) \\
\mathit{receive\_server} & ::= & \textbf{? *[}\ \sum_i M_i\textbf{;}\ I_i\ \textbf{]} \\
\mathit{receive} & ::= & \textbf{? [}\ \sum_i M_i\textbf{;}\ I_i\ \textbf{]} \\
\mathit{send} & ::= & \textbf{! [}\ \sum_i M_i\textbf{;}\ I_i\ \textbf{]} \\
I & ::= & \textbf{0} \mid \mathit{peer\_name} \mid \mathit{mod}\ \mathit{send} \mid \mathit{mod}\ \mathit{receive} \\
\mathit{mod} & ::= & \textbf{may} \mid \textbf{must} \\
M & ::= & \mathit{name}[\ \textbf{(}\ \widetilde{\mathit{args}}\ \textbf{)}\ ] \\
\mathit{args} & ::= & \mathit{basic\_type} \mid \mathit{peer\_name} \mid \mathit{server\_name} \\
\mathit{basic\_type} & ::= & \textbf{boolean} \mid \textbf{integer} \mid \textbf{real} \mid \textbf{string}.
\end{array}
$$

The **!** and **?** keywords are the usual sending and receiving actions. The choice operator **+** allows to choose one message among the list[1], and the **;** is used to sequence behaviors. The ***** keyword allows specification of servers, and is discussed later in this section. **0** (zero) stands for the termination of the interface. The modalities **may** and **must** distinguish between permissions and obligations for the performance of the actions. Their meaning is (where the partner is the other communicating port):

may ? [ΣM_i**;** I_i **]**	"the port does not impose any sending constraint on the partner, but if the partner sends any message M_i, then the port guarantees to be ready to receive it, and to execute later on action I_i".
must ? [ΣM_i**;** I_i **]**	"the port does impose a sending constraint on the partner, and if the partner sends any message M_i, then the port guarantees to be ready to receive it, and to execute later on action I_i".
may ! [ΣM_i**;** I_i **]**	"the port may send one of the messages M_i to its partner and execute later on action I_i; the partner must be ready to receive the message M_i".
must ! [ΣM_i**;** I_i **]**	"the port guarantees to send one of the messages M_i to its partner and execute later on action I_i; the partner must be ready to receive the message M_i".

Messages contain arguments. Thus, basic data (such as integers, booleans...) as well as references to ports (be it *peer_name* or *server_name*), can be passed in messages. Our type language does not cater for structured data, but their addition is straightforward. Sending or receiving references implies some restrictions that are enforced on the behaviour of the involved components:

! m (I)	"the port is sending to its partner a reference to a port whose behavior is described by the type I. Moreover, the first action of this referenced port must be **?** (reception)."
? m (I)	"the port is receiving a reference to another port whose behavior is conform to the type I. Moreover, the first action of this referenced port is a **?**."

The constraint on the first action of the referenced port is inevitable: if the first action of I is **!**, then the referenced port can send this message to a third port, which will lead to incompatible behaviours between components.

Finally, the *****-construct allows specification of a server. This server spawns to answer a request, so it immediately reconfigures to honour other potential clients:

$I = mod$ **? *[** $m()$**;** I' **]**	after the reception of m, a port whose behaviour is I' is created while the server is regenerated as I. The new port will interact with the sender of the request. The first action of I' must be **!** (send), so the client will be informed of the port created for its request.

Several client ports can be connected at the same time to the same server port. As for peer ports, they are connected only in pairs. This distinction allows client-server and peer-to-peer links.

1. We consider that all messages of this list are distinct.

As regards to server types, we remark that a type (peer or server) cannot become a type declared as server. For example, the following declarations are forbidden (type I becomes server type $Server$; type $Server$ becomes server type $Server$):

$$I \textbf{ = may ! [} M\textbf{; } Server \textbf{]} \qquad Server \textbf{ = may ? *[} M\textbf{; } Server \textbf{]}$$

Our type language has the effect of not only imposing constraints on the component, but also of imposing constraints on its partner (i.e. on the environment). The introduction of modalities leads to an underlying model which is a kind of *modal* $\mathcal{LTS}$, in which states can be either **may** or **must** [LSW95] . This has a strong impact on the type compatibility rules, which are discussed in Section 2.3

2.2. *Subtyping*

In this section, we define a subtype relation, which has the classic property of subtypes (meaning the subtype can replace the supertype). The main idea of T being a subtype of S, noted $T \preceq S$, is the following:

– If T and S are receiving, then:
  - T receives *at least* the messages S can receive;
  - the subtype relation is *contra-variant*;
  - if S has **may** modality, then T has **may** modality. Otherwise there is no restriction.

– If T and S are sending, then:
  - T sends *at most* the messages S can send;
  - the subtype relation is *co-variant*;
  - if S has **must** modality, then T has **must** modality. Otherwise there is no restriction.

For example, with a lightened syntax, we have:

$$\begin{array}{rcl}
\textbf{may ?}\ M_1 + M_2 + M_3 & \preceq & \textbf{may ?}\ M_1 + M_2 \\
(\textbf{may}|\textbf{must})\ \textbf{?}\ M_1 + M_2 + M_3 & \preceq & \textbf{must ?}\ M_1 + M_2 \\
(\textbf{may}|\textbf{must})\ \textbf{!}\ M_1 & \preceq & \textbf{may !}\ M_1 + M_2 \\
\textbf{must !}\ M_1 & \preceq & \textbf{must !}\ M_1 + M_2
\end{array}$$

We further explain our choice on the subtyping of modalities. When receiving, if the supertype is **may?**, it does not impose any constraint on its partner. Hence the subtype cannot impose any constraint either, which leads to the **may?** subtype. Similar reasoning explains other subtype relations.

The subtype relation is such that it also recursively applies to the type of the actions following reception and sending: if $I = \textbf{may ! [} M_1\textbf{;} I_1 \textbf{]}$ is a subtype of $J = \textbf{may ! [} M_1\textbf{;} J_1 \textbf{ + } M_2\textbf{;} J_2 \textbf{]}$ then I_1 is a subtype of J_1.

The subtype relation is formally described by the rules in Tab. 1 and the subtype predicate $\preceq_m$ on modalities. Three cases in this predicate deserve a few lines.

$T_{\mathrm{mod}} \preceq_m S_{\mathrm{mod}}$

T_{mod} \ S_{mod}	**must?**	**may?**	**must!**	**may!**	**0**
must?	$\surd$				
may?	$\surd$	$\surd$		$\surd$	$\surd$
must!			$\surd$	$\surd$	
may!				$\surd$	
0				$\surd$	$\surd$

$$\text{ST-BOOL}\ \frac{}{\Gamma \vdash \textbf{boolean} \preceq \textbf{boolean}} \qquad \text{ST-ASSUMP}\ \frac{I \preceq J \in \Gamma}{\Gamma \vdash I \preceq J}$$

$$\text{ST-STRING}\ \frac{}{\Gamma \vdash \textbf{string} \preceq \textbf{string}} \qquad \text{ST-NAME1}\ \frac{\Gamma, name \preceq I \vdash \mathrm{replace}(name) \preceq I}{\Gamma \vdash name \preceq I}$$

$$\text{ST-INT}\ \frac{}{\Gamma \vdash \textbf{integer} \preceq \textbf{integer}} \qquad \text{ST-NAME2}\ \frac{\Gamma, I \preceq name \vdash I \preceq \mathrm{replace}(name)}{\Gamma \vdash I \preceq name}$$

$$\text{ST-INT2}\ \frac{}{\Gamma \vdash \textbf{integer} \preceq \textbf{real}}$$

$$\text{ST-MAYRECV}\ \frac{}{\Gamma \vdash \textbf{may?}\ [\textbf{*}][\textbf{ }\Sigma_i M_i(\tilde{I}_i)\textbf{; } T_i\textbf{ }] \preceq \textbf{0}}$$

$$\text{ST-REAL}\ \frac{}{\Gamma \vdash \textbf{real} \preceq \textbf{real}}$$

$$\text{ST-MAYSEND}\ \frac{}{\Gamma \vdash \textbf{0} \preceq \textbf{may !}\ [\textbf{ }\Sigma_i M_i(\tilde{I}_i)\textbf{; } T_i\textbf{ }]}$$

$$\text{ST-ZERO}\ \frac{}{\Gamma \vdash \textbf{0} \preceq \textbf{0}}$$

$$\text{ST-RECVSEND}\ \frac{}{\Gamma \vdash \textbf{may ?}\ [\textbf{*}][\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant n} N_i(\tilde{J}_i)\textbf{; } T_i\textbf{ }] \preceq \textbf{may !}\ [\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textbf{; } S_i\textbf{ }]}$$

$$\text{ST-RECV}\ \frac{\forall i \in \{1..m\} : \Gamma \vdash \tilde{I}_i \preceq \tilde{J}_i \ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash mod_T\ \textbf{?}\ [\textbf{*}][\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant n} M_i(\tilde{J}_i)\textbf{; } T_i\textbf{ }] \preceq mod_S\ \textbf{?}\ [\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textbf{; } S_i\textbf{ }]}\ \Box$$

$$\text{ST-RECV*}\ \frac{\forall i \in \{1..m\} : \Gamma \vdash \tilde{I}_i \preceq \tilde{J}_i \ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash mod_T\ \textbf{?*}[\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant n} M_i(\tilde{J}_i)\textbf{; } T_i\textbf{ }] \preceq mod_S\ \textbf{?*}[\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textbf{; } S_i\textbf{ }]}\ \Box$$

$$\text{ST-SEND}\ \frac{\forall i \in \{1..n\} : \Gamma \vdash \tilde{J}_i \preceq \tilde{I}_i \ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash mod_T\ \textbf{!}[\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant n} M_i(\tilde{J}_i)\textbf{; } T_i\textbf{ }] \preceq mod_S\ \textbf{!}[\textbf{ }\textstyle\sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textbf{; } S_i\textbf{ }]}\ \Diamond$$

$$\Box \triangleq (mod_T\ \textbf{?} \preceq_m mod_S\ \textbf{?}) \ \wedge\ n \geqslant m$$

$$\Diamond \triangleq (mod_T\ \textbf{!} \preceq_m mod_S\ \textbf{!}) \ \wedge\ n \leqslant m$$

$$\tilde{I} \preceq \tilde{J} \triangleq \tilde{I} = (I_i)_{1..k}, \tilde{J} = (J_i)_{1..k} \ \wedge\ \forall i, I_i \preceq J_i$$

Table 1. *Subtyping rules* ([*****] means the *****-construction is optional)

may? $\preceq_m$ **0** and **0** $\preceq_m$ **may!** illustrate the semantics of **may** modality: **may?** can replace an inactive interface **0** (no message will be sent to the inactive interface **0**, hence no message will be sent to the **may?** interface). Also, an inactive interface **0** can be considered as an interface that can send messages (but with no obligation). Finally, because subtype relations are transitive, we have another relation: **may?** $\preceq_m$ **may!**. At first sight, this relation seems strange; however, by looking at the semantics of modalities, we have a supertype which *may* send, and a subtype **may?** which will never send messages, thus respecting the **may!** semantics. This relation is surprising especially because one is used to conceive sendings and receivings with the **must** modality.

Once the principles of the $\preceq_m$ relation are understood, subtying rules are quite straightforward. The first six ones (on the left column) deal with basic types and inactive interfaces. Rules ST-ASSUMP, ST-NAME1 and ST-NAME2 take into account recursive interfaces, and cases where interface is pointed out using its name ($\mathrm{replace}(\textit{name})$ replaces *name* with the definition of the corresponding type). Rules ST-MAYRECV and ST-MAYSEND concern subtyping staking type **0**. The rule ST-RECVSEND tackles the **may?** $\preceq_m$ **may!** relation: note the message lists between sub- and super-type are different. The three last rules give the classical definition of interface subtypes: the subtype can receive more, and send less than its supertype. Order of messages is not considered of importance (as all messages are distinct).

Concerning server interfaces, a subtype can be a server (while the supertype is not), but the opposite is forbidden, as shown with the rule ST-RECV*: this rule is identical to ST-RECV, but applies to server types (both sub- and supertypes). This restriction is due to the fact that server ports can be connected to several ports: if the supertype $S^{\star}$ is a server port, then this port can answer several requests, which will not be the case of a port whose subtype $T \preceq S^{\star}$ is not a server. On the contrary, $T^{\star} \preceq S$ does not raise this kind of problems.

Property 1
$\preceq$ is a partial preorder.

Proof. Truth table of $\preceq_m$ assures transitivity and reflexivity among modalities. The rest of the demonstration is made by structural induction. □

2.3. *Compatibility Rules*

In this section we define the symmetric predicate$Comp(I, J)$ as "I and J are compatible with each other". Compatibility between interfaces I and J is informally defined as follows:

$I =$ **must ?** m implies $J =$ **must !** m
$I =$ **may ?** m implies $J =$ **must !** m or $J =$ **may !** m or $J =$ **0** or $J =$ **may ?** m'
$I =$ **must !** m implies $J =$ **must ?** m or $J =$ **may ?** m
$I =$ **may !** m implies $J =$ **may ?** m
$I =$ **0** implies $J =$ **may ?** m or $J =$ **0**

The compatibility rules are actually defined using several elementary compatibility relations: compatibility between modalities, messages, and finally types.

We first define the compatibility between modalities, as the symmetric boolean relation $Comp_{\text{mod}}(mod_I\ [\textbf{!}|\textbf{?}], mod_J\ [\textbf{!}|\textbf{?}])$. Its truth table is reproduced hereafter:

	I				
J	**must?**	**may?**	**must!**	**may!**	**0**
must?			$\surd$		
may?		$\surd$	$\surd$	$\surd$	$\surd$
must!	$\surd$	$\surd$			
may!		$\surd$			
0		$\surd$			$\surd$

We defi ne also$Comp_{\text{msg}}$, a relation over message types. Two message types are compatible iff they have the same name and their arguments are pairwise related such that arguments of sent messages are subtypes of arguments of received messages. This is formally defi ned (with $M_{\mathbf{!}}$ the sent message, and $M_{\mathbf{?}}$ the received one):

$$\begin{aligned} Comp_{\text{msg}}(M_{\mathbf{!}}, M_{\mathbf{?}}) &\triangleq Comp_{\text{msg}}(M_{\mathbf{!}}(I_1, \ldots, I_n), M_{\mathbf{?}}(J_1, \ldots, J_n)) \\ &\triangleq M_{\mathbf{!}} = M_{\mathbf{?}} \ \wedge \ \forall i, I_i \preceq J_i \end{aligned}$$

We can then defi ne the compatibility $Comp(I, J)$ between two interfaces as compatibility between modalities and messages, and transitions must lead to compatible interfaces. This is formally defi ned recursively as (with $\rho \in \{\mathbf{?}, \mathbf{!}\}$, and where [*****] means that the *****-construct may be present or not):

$$\begin{aligned}
Comp(I, J) &\triangleq Comp(J, I) \\
Comp(\mathbf{0}, \mathbf{0}) &\triangleq true \\
Comp(\mathbf{0}, mod_J\ \rho_J\ [\mathbf{*}][\ \Sigma_l\ M'_l\mathbf{;}\ J_l\ ]\ ) &\triangleq Comp_{\text{mod}}(\mathbf{0}, mod_J\ \rho_J) \\
Comp(\mathbf{may?}\ [\mathbf{*}][\ \Sigma_l\ M_k\mathbf{;}\ I_k\ ],\ \mathbf{may?}\ [\mathbf{*}][\ \Sigma_l\ M'_l\mathbf{;}\ J_l\ ]) &\triangleq true \\
Comp(mod_I\ \mathbf{!}\ [\ \Sigma_k\ M_k\mathbf{;}\ I_k\ ],\ mod_J\ \mathbf{?}\ [\mathbf{*}][\ \Sigma_l\ M'_l\mathbf{;}\ J_l\ ]\ ) &\triangleq Comp_{\text{mod}}(mod_I\ \mathbf{!}, mod_J\ \mathbf{?}) \\
&\quad \wedge \ \big(\forall k, \exists l : Comp_{\text{msg}}(M_k, M'_l) \wedge Comp(I_k, J_l)\big)
\end{aligned}$$

The recursive defi nition indicates that the compatibility of a pair of interfaces is a boolean function of a fi nite set of pairs of interfaces. This defi nition also closely resembles the defi nition of simulation or equivalence relations over fi nite state transition systems. Hence, the verifi cation of compatibility always terminates, and can be performed with standard techniques in a quadratic complexity with the number of interfaces (intended as different states of the interfaces). Due to the abstraction used in the defi nition of interfaces, such number is small with respect to the complexity of the component behaviour. Moreover, the wide range of techniques introduced for the effi cient verifi cation of fi nite state systems can be studied in search of the ones that best fi t this specifi c verifi cation problem.

2.4. *Properties of the Subtypes*

In this section, we prove the classical property of subtypes, i.e. that a subtype can replace supertype without any change of the partner of the supertype:

Property 2
If $T \preceq S$ *then* $\forall I, Comp(I,S)$ *implies* $Comp(I,T)$

Lemma 1 (subtyping and modalities)
If $T_{\mathrm{mod}} \preceq_m S_{\mathrm{mod}}$ *then* $\forall I_{\mathrm{mod}}, Comp_{\mathrm{mod}}(I_{\mathrm{mod}}, S_{\mathrm{mod}})$ *implies* $Comp_{\mathrm{mod}}(I_{\mathrm{mod}}, T_{\mathrm{mod}})$.

Proof. Straightforward from truth tables of $\preceq_m$ and $Comp_{\mathrm{mod}}$ (by looking only at cases where T and S modalities are different, there are 5 cases). □

Proof (prop. 2).
By induction on the structure of the types. We omit the case where S is a server, but proof is identical. Given T, S and I such that $T \preceq S$ and $Comp(I,S)$. There is three cases:

1) S is receiving.

Let's write $S = mod_S$ **? [** $\sum_{1 \leqslant l \leqslant m} M_l^S(\tilde{J}_l^S)$**;** S_l **]**. Receiving supertypes are concerned only by rules ST-RECV, ST-RECV*, which give $T = mod_T$ **?** [*]**[** $\sum_{1 \leqslant l \leqslant n} M_l^T(\tilde{J}_l^T)$**;** T_l **]** with: mod_T **?** $\preceq_m mod_S$ **?**, $n \geqslant m$ and $\forall l \in \{1..m\} : \tilde{J}_l^S \preceq \tilde{J}_l^T,\ T_l \preceq S_l,\ M_l^T = M_l^S$.

Compatibility between I and S has the following cases:

$I =$ **0** or $I =$ **may?** [*]**[** $\sum_k M_k^I(\tilde{J}_k^I)$**;** I_k **]**.
Then we have $Comp(I,T) = Comp_{\mathrm{mod}}(mod_I, mod_T$ **?**$) =$ true, by lemma 1.

$I = mod_I$ **! [** $\sum_k M_k^I(\tilde{J}_k^I)$**;** I_k **]**.
Then $Comp(I,S)$ is written:
$Comp_{\mathrm{mod}}(mod_I$ **!**$, mod_S$ **?**$)\ \wedge\ \forall k, \exists l : Comp_{\mathrm{msg}}(M_k^I, M_l^S) \wedge Comp(I_k, S_l)$.
By lemma 1, the left term of the conjunction implies $Comp(mod_I$**!**$, mod_T$**?**$)$. As for the right term, we have by definition $Comp_{\mathrm{msg}}(M_k^I, M_l^S) \triangleq M_k^I = M_l^S \wedge \tilde{J}_k^I \preceq \tilde{J}_l^S \triangleq M_k^I = M_l^T \wedge \tilde{J}_k^I \preceq \tilde{J}_l^T$ by definition of $T \preceq S$ and transitivity of $\preceq$. Moreover, as $Comp(I_k, S_l)$ is true, we have, by induction, $T_l \preceq S_l$, and by lemma 1: $Comp(I_k, T_l)$ is true, which leads to the conclusion.

2) S is sending (the reasoning is similar to the previous case).

Cases where T is not sending are trivial: S has then **may!** modality, thus I has **may?** modality which is compatible with **0** and **may?**, hence with T.

Suppose T is sending, and let's write $S = mod_S$ **! [** $\sum_{1 \leqslant l \leqslant m} M_l^S(\tilde{J}_l^S)$**;**$S_l$**]** and $T =$ mod_T **! [** $\sum_{1 \leqslant l \leqslant n} M_l^T(\tilde{J}_l^T)$**;**$T_l$**]**, with $n \leqslant m$, mod_T **!** $\preceq_m mod_S$ **!** and
$\forall i \in \{1..n\} : \tilde{J}_i^T \preceq \tilde{J}_i^S \wedge T_i \preceq S_i \wedge M_i^T = M_i^S$. I has to be receiving:
$I = mod_I$ **? [** $\sum_k M_k^I(\tilde{J}_k^I)$**;** I_k **]**. Compatibility between I and S give:
$Comp_{\mathrm{mod}}(mod_S$ **!**$, mod_I$ **?**$) \wedge \forall k \in \{1..m\}, \exists l : Comp_{\mathrm{msg}}(M_k^S, M_l^I) \wedge Comp(S_k, I_l)$.
Particularly: $\forall k \leqslant n \leqslant m, \exists l : Comp_{\mathrm{msg}}(M_k^S, M_l^I) \wedge Comp(S_k, I_l)$. As $M_k^T = M_k^S$, $\tilde{J}_k^S \preceq \tilde{J}_k^T$ and $T_k \preceq S_k$, we are sure that, for any message from T, there exists a message from I satisfying $M_k^T = M_l^I \wedge \tilde{J}_k^T \preceq \tilde{J}_l^I \wedge Comp(T_k, I_l)$. Finally lemma 1 allows to conclude.

3) S has no action ($S =$ **0**).

Then T has **may?** modality, and I has either **may?** modality, or is inactive (**0**). $Comp(I,T)$ is straightforward. □

2.5. *Dual Interfaces*

We define a duality relation, noted $^{\mathcal{D}}$, for peer interfaces, such that $I^{\mathcal{D}}$ is unique and compatible with I. The dual of I is simply calculated by exchanging sendings and receivings:

$$\mathbf{0}^{\mathcal{D}} \triangleq \mathbf{0}$$

$$\left(mod\ \mathbf{!}\ [\ \Sigma_i M_i(\tilde{J}_i);\ I_i\ ]\right)^{\mathcal{D}} \triangleq mod\ \mathbf{?}\ [\ \Sigma_i M_i(\tilde{J}_i);\ {I_i}^{\mathcal{D}}\ ]$$

$$\left(mod\ \mathbf{?}\ [\ \Sigma_i M_i(\tilde{J}_i);\ I_i\ ]\right)^{\mathcal{D}} \triangleq mod\ \mathbf{!}\ [\ \Sigma_i M_i(\tilde{J}_i);\ {I_i}^{\mathcal{D}}\ ]$$

The dual has the following properties (where the first three ones are obvious):

Property 3 (dual)

3.1) $I^{\mathcal{D}}$ *is unique;*

3.2) $\left(I^{\mathcal{D}}\right)^{\mathcal{D}} = I$*;*

3.3) $Comp(I^{\mathcal{D}}, I)$*;*

3.4) $Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$*: the greater supertype compatible with* J *is* $J^{\mathcal{D}}$*;*

3.5) $I \preceq J \Leftrightarrow J^{\mathcal{D}} \preceq I^{\mathcal{D}}$

Proof (prop. 3.4).
1.$\Rightarrow$: Proof is similar to the one of property 2. Sketch is the following.
Suppose I is sending. We have:
$I = mod_I\mathbf{![}\Sigma_k M_k^I(\tilde{I}'_k)\mathbf{;}I_k\mathbf{]} \quad J = mod_J\mathbf{?[}\Sigma_l M_l^J(\tilde{J}'_l)\mathbf{;}J_l\mathbf{]} \quad J^{\mathcal{D}} = mod_J\mathbf{![}\Sigma_l M_l^J(\tilde{J}'_l)\mathbf{;}{J_l}^{\mathcal{D}}\mathbf{]}$
The definition of compatibility give:
$Comp_{\text{mod}}(mod_I\mathbf{!}, mod_J\mathbf{?}) \quad \text{and} \quad \forall k, \exists l : M_l^I = M_k^J \wedge \tilde{I}'_k \preceq \tilde{J}'_l \wedge Comp(I_k, J_l)$
From which we deduce $mod_I\mathbf{!} \preceq mod_J\mathbf{!}$ and, with a rearrangement of messages:
$mod_I\mathbf{![}\Sigma_k M_k^I(\tilde{I}'_k)\mathbf{;}\ I_k\ \mathbf{]} \preceq mod_J\mathbf{![}\Sigma_l M_l^J(\tilde{J}'_l)\mathbf{;}\ J_l\ \mathbf{]}$ with $\tilde{I}'_k \preceq \tilde{J}'_l \wedge Comp(I_k, J_l)$
With the induction hypothesis, we have $I_k \preceq {J_l}^{\mathcal{D}}$. We conclude $I \preceq J^{\mathcal{D}}$.

Case $I = \mathbf{0}$ is similar: J is either $\mathbf{0}$, or **may ? [**$\ldots$**]**, the dual of which is supertype of $\mathbf{0}$. Case I receiving is dealt the same as I sending: case J sending is identical. When J has **may?** modality, then I has the same modality, and the subtype relation we are looking for is **may?** $\preceq$ **may!**, which is true. Case $J = \mathbf{0}$ and $I =$ **may?[**$\ldots$**]** leads to $I \preceq J^{\mathcal{D}}$.

2. $\Leftarrow$:
We have $Comp(J^{\mathcal{D}}, J)$ (prop. 3.3), from which we deduce $Comp(I, J)$ (prop. 2). □

Proof (prop. 3.5).
$I \preceq J \Leftrightarrow Comp(I, J^{\mathcal{D}}) \Leftrightarrow J^{\mathcal{D}} \preceq I^{\mathcal{D}}$ □

2.5.1. *Duality and Server Types*

Our dual function can be extended to server types as follows (where the dual loses the *****):

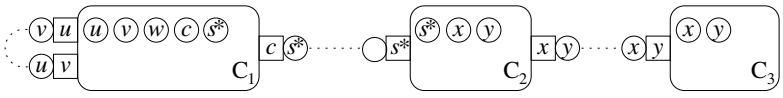
$$\left(mod\ \mathbf{?*[}\Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ I_i\ \mathbf{]}\right)^{\mathcal{D}} \triangleq mod\ \mathbf{!}\ \mathbf{[}\Sigma_i M_i(\tilde{J}_i)\mathbf{;}\ {I_i}^{\mathcal{D}}\ \mathbf{]}$$

The dual is unique, but we no longer have, for a server I: $\left(I^{\mathcal{D}}\right)^{\mathcal{D}} = I$. However, other properties hold, especially $Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$.

3. Component Model

3.1. *Informal Presentation*

Our computational model describes a system as a confi guration of communicating components. Each component owns a set of ports, and communication is by asynchronous message passing between ports. Sending from a port can only occur if this port is bound to another "partner" port; then, any message sent from it is routed to this partner port. An unbound port can only perform receptions (this is the typical case for server ports). We consider dynamic confi gurations: a component may create new ports and may also dynamically bind a partner reference to any of its owned ports. In our setting, both peer to peer and client/server communications can be modelled: when two ports are mutually bound, they are peers; when the binding is asymmetrical, the bound port is a client and the unbound port is its server. We constrain peer references/ports to be private [NNS99], meaning a peer port is known only of its partner. Figure 1 shows a confi guration made of three components. Note how port c (in C_1) is asymmetrically bound to s (in C_2; flag * indicates that reference s is a server), and the peer to peer binding between ports x and y (in C_2, C_3), and between u and v (both in C_1).



$$\mathsf{R}_1 = \{u, v, w, c, s^*\},\ \mathsf{P}_1 = \{(u \multimap v), (v \multimap u), (c \multimap s^*)\}$$

$$\mathsf{R}_2 = \{s^*, x, y\}\ \mathsf{P}_2 = \{(s^* \multimap \bot), (x \multimap y)\} \qquad \mathsf{R}_3 = \{x, y\}\ \mathsf{P}_3 = \{(y \multimap x)\}$$

Figure 1. *An example of a confi guration*

Components are also multi-threaded. We consider here an abstract thread model, focusing only on external, port based, manifestations of threads. Thus, an active thread is a chain made of a head port (the active port), and a tail (the ordered sequence of suspended ports). The thread chain may dynamically grow or decrease: this happens when the head port is suspended and the activity is given to another port, and when the head port is removed from the chain (because it terminated or became idle) and the port next to the head becomes active.

Since in this paper we focus on the interface typing issues, we do not provide a fully-fledged syntax for components. Rather, we defi ne an abstract behavioural model of components in terms of their observable transitions and their multi-threaded, port-located, activities. The abstract model defi ned in this section is general and independent of any concrete behavioural notation for components.

3.2. *Notations for Components*

A component is made of a state, a set of ports, a set of references, and a set of threads.

The set of references is noted R, and is ranged over by u, v, w, c, s^*. The *-mark points out a reference to a server port. Classical set notation is used for the operations on R; however, we use the unorthodox $R \cup u$ notation for the insertion of element u into R, without duplicates.

The set of ports is noted P. It is a set of mappings from ports to partner references. Elements of P are noted $(\cdot \multimap \cdot)$. If a port u is bound to a partner v, then $(u \multimap v) \in \mathsf{P}$. If u has no partner, then $(u \multimap \bot) \in \mathsf{P}$. We also write $u \in \mathsf{P}$ if u is a local port belonging to P.

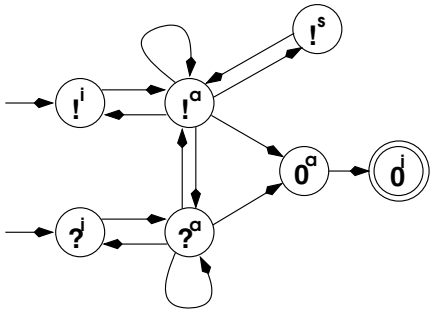
The following operators will be useful for the manipulation of port mappings:

$\mathsf{P}[u \multimap \bot]$	port u is added to P.
$\mathsf{P}[u \multimap v]$	bind the partner v to port u. Overrides the previous partner.
$\mathsf{P}\backslash u$	remove the port u from P.

The set of threads, T, reflects the state of the ports of the component and the dependencies between them. A port u depends (or suspends) on a port v when an action from u was not terminated, and this action needs some interaction from v (with v's partner). For example, in a proxy component, if u receives a request from a client, it will suspend to v which will forward the request to the server and recover the answer; then u will become active again and answer back to its client. The status of a port is abstracted from its activity (activated, suspended or idle) and from its state (sending, receiving or no action). We formally denote the activity state of a port u as $u\rho^\sigma$, with activity ρ and state σ such that:

$$\rho = \begin{cases} \mathbf{!} & u \text{ is in a sending state} \\ \mathbf{?} & u \text{ is in a receiving state} \\ \mathbf{0} & u \text{ has no action} \end{cases}$$

$$\sigma = \begin{cases} \mathsf{a} & u \text{ is active} \\ \mathsf{s} & u \text{ is suspended (by a port)} \\ \mathsf{i} & u \text{ is idle} \end{cases}$$



The state diagram shows the possible evolutions of the activity of a port. For example, $u\mathbf{!}^{\mathsf{a}}$ shows that the port u is active, and its next action is sending. This port can either send its message, be suspended by another port, or become idle. Some restrictions apply:

– when a port is created, it is in an idle state;

– $\mathbf{0}$ is terminal. $u\mathbf{0}^{\mathsf{a}}$ can only give back the thread of control, become $u\mathbf{0}^{\mathsf{i}}$ and vanish;

– $u\mathbf{!}^{\mathsf{a}}$ is an active port waiting to send a message. It can either send the message and change its state, suspend to another port, or become idle;

– $u\mathbf{?}^{\mathsf{a}}$ is an active port waiting for an incoming message. It can either receive the message and change its state, or become idle[2];

– only the sending ports can be suspended: combination $u\mathbf{?}^{\mathsf{s}}$ is forbidden.

We let x, y range over port activity states. We use the notation $x \rightarrowtail y$ which denotes x is suspended by y; this means that the activation of x is pending until y terminates (y has no action) or passivates (y becomes idle).

$\mathsf{T} = t_1 | \cdots | t_n$ is a set of parallel threads where a thread t is a sequence $x_1 \rightarrowtail x_2 \rightarrowtail \cdots \rightarrowtail x_n$. We note $|t|$ the length of thread t. This sequence has the following constraints:

2. However, a receiving port can become idle only if it will no longer receive messages (see Section 4).

$x_i = u_i!^{\mathsf{s}}$ iff $i < |t|$ (all the ports but the last one are suspended)
$n = |t| > 1 \Rightarrow x_n = u_n\rho_n^{\mathsf{a}}$ (a sequence of more than one port ends with an active port)

In this paper, our model does not take into account dependencies between threads, for the sake of space. However, this extension is fully developed in [Car03].

The following operations on T are defined. We add a pool of idle ports: in the rules, ports are created and removed from this pool. Each port occurs only once in T.

$\mathsf{T} \mid u\rho^{\mathsf{i}}$	add a port in the idle ports pool.
$\mathsf{T}\backslash u$	remove u from T and activate the port that was possibly suspended by u. This operation is defined only if u is the head of some thread $t \in \mathsf{T}$.
$\mathsf{T}[u \rightarrowtail v]$	suspends port u to port v. This operation is defined only if u is the head of some thread $t_1 \in \mathsf{T}$ and v is in a singleton thread $t_2 = v\rho^{\mathsf{i}} \in \mathsf{T}$. It changes the state of u to suspended, adds the new head $v\rho^{\mathsf{a}}$ to t_1, and removes t_2. Note that a port can be head of only one thread at a time.
$\mathsf{T}[u\rho'/u\rho]$	modifies the state of a port in P: only ρ changes to ρ'.
$\mathsf{T}[u\rho^{\sigma\rightarrow\sigma'}]$	changes the activity of a port from σ to σ'.
$\mathsf{T}(u)$	returns ρ^σ if $u\rho^\sigma \in \mathsf{T}$.

3.3. *Communication Medium*

As indicated in the introduction, communication between components is by asynchronous message passing. Thus, a message is first deposited by its sender into a communication medium and, in a later stage, removed from this medium by its receiver. The delivery discipline that we adopt is first in first out. We define Com as a communication abstraction containing a collection of fifo queues, one for each port in the component: messages are written to and read from Com. We define the following notation on Com:

$Com[\triangleleft u]$	inserts a new queue for port u.
$Com.u$	the queue for port u. It is an ordered set of messages «$v : M(\tilde{w})$» where v is the reference of the sending port, M is the name of the message, and $\tilde{w}$ its arguments. If u is not defined, $Com.u = \bot$.
$Com\backslash u$	the queue u is removed.
$Com[u\triangleright]$	remove from the queue associated with u the next message.
$Com[u \triangleleft v : M(\tilde{w})]$	put message «$v : M(\tilde{w})$» in the queue associated with u.
$Com.u_\triangleright$	yields the next message (in queue u) to be treated.

3.4. *Component Semantics*

A component is defined by its state and its set of ports, references and threads:

$B(\mathsf{P}, \mathsf{R}, \mathsf{T})$ where B is the state of the component;
P, R, T are the ports, references and threads as defined previously.

The rules in Tab. 3 describe the semantics for the components, showing the transitions a component may perform in a given communication abstraction. A transition may change the state of the component itself and/or that of the communication abstraction. The actions of the

component deal essentially with handling of ports (creation/removal), references (bindings, removals), threads (activate/suspend) and messages (send/receive).

C-CREAT allows creation of a port in the component. The rule adds an unbounded and idle port, a reference to this port (so it can possibly be sent), and a corresponding queue in Com.
C-REMVPORT removes a port. It occurs only if the port is idle and its queue is empty.
C-REMVREF removes a reference from R. This reference must neither be a port ($u \notin \mathsf{P}$), nor bounded to a port ($u \notin CoDom(\mathsf{P})$).
C-BIND deals with binding of a partner. It is allowed only if the port has no partner ($(u \multimap \bot)$), if the port is sending, and not suspended ($\mathsf{T}(u) = \mathbf{!}^{\mathsf{a,i}}$)[3]. If the partner is a peer reference, it is bound to only one port at a time ($\mathrm{peer}(v) \Rightarrow v \notin CoDom(\mathsf{P})$).
C-ACTV allows the port u to suspend on a port v. u must be active, in a sending state ($\mathsf{T}(u) = \mathbf{!}^{\mathsf{a}}$), and v idle, in a sending state ($\mathsf{T}(v) = \mathbf{!}^{\mathsf{i}}$).
C-ACTV2 activates a port (from the pool of idle ports) with its own thread.
C-DEACT deactivates a port; this port has to be active.
C-SEND deals with sendings. The port must be active, in a sending state, bound to a reference ($(u \multimap u')$). Sent peer references must not be bound to a port ($\mathrm{peer}(\tilde{v}) \cap CoDom(\mathsf{P}) = \varnothing$), and must be removed from R ($\mathsf{R}' = \mathsf{R} - \mathrm{peer}(\tilde{v} \cup \{u\})$); as the reference to the port is also sent, it is removed from R if it is a peer. This ensures that peer references are known from only one component. After the sending, the port changes to the next action ($\mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{!}]$).
C-RECV deals with receivings. The sender becomes the new partner ($\mathsf{P}' = \mathsf{P}[u \multimap u'']$, only if u is a peer port), and the old one is removed ($\{u' | (u \multimap u')\}$ removed from R); this allows the partner to delegate its behaviour to another port. Received references and the new partner are stored in R. After the reception, the port changes to the next action ($\mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{?}]$).

3.5. *Configuration of Components*

When we take into account a confi guration made up of several components, we consider the communication medium Com as shared among the components. This way, queues are shared and components can communicate through them, asynchronously. We give in Tab. 2 the communication rule for a confi guration with two components, illustrating the sharing of Com. Extension to confi gurations with more components is straightforward.

$$\text{CPAR}\ \frac{B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1), Com \xrightarrow{\alpha} B_1'(\mathsf{P}_1', \mathsf{R}_1', \mathsf{T}_1'), Com'}{B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1) \mid B_2(\mathsf{P}_2, \mathsf{R}_2, \mathsf{T}_2), Com \xrightarrow{\alpha} B_1'(\mathsf{P}_1', \mathsf{R}_1', \mathsf{T}_1') \mid B_2(\mathsf{P}_2, \mathsf{R}_2, \mathsf{T}_2), Com'}$$

Table 2. *Rules for confi gurations of components*

4. Contract Satisfaction

The interface language presented in Section 2 imposes constraints on the remote interface, which will imply constraints also on the components. In this section, we present typing relation between components and the interface language, so the component will respect a

3. Binding a reference to a receiving port is useless as it will be erased during the reception (cf. rule C-RECV) and a binding when the port is suspended can be differed until the port is active.

$$\text{C-CREAT}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap \bot] \qquad \mathsf{R}' = \mathsf{R} \cup u \qquad \mathsf{T}' = \mathsf{T} \mid u\rho^{\mathsf{i}} \qquad Com' = Com[\triangleleft u]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\mathrm{creat}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ u \notin \mathsf{P} \ \wedge\ Com.u = \bot$$

$$\text{C-REMVPORT}\ \frac{\mathsf{P}' = \mathsf{P}\backslash u \qquad \mathsf{R}' = \mathsf{R}\backslash u \qquad \mathsf{T}' = \mathsf{T}\backslash u \qquad Com' = Com\backslash u}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\mathrm{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ \mathsf{T}(u) = \rho^{\mathsf{i}} \ \wedge\ Com.u = \varnothing$$

$$\text{C-REMVREF}\ \frac{\mathsf{R}' = \mathsf{R}\backslash u}{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}\ u \notin \mathsf{P} \cup CoDom(\mathsf{P})$$

$$\text{C-BIND}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\mathrm{bind}(u \multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T}), Com}\ \square$$

$$\text{C-ACTV}\ \frac{\mathsf{T}' = \mathsf{T}[u \rightarrowtail v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\mathrm{actv}(u \rightarrowtail v)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \mathbf{!}^{\mathsf{a}} \ \wedge\ \mathsf{T}(v) = \mathbf{!}^{\mathsf{i}}$$

$$\text{C-ACTV2}\ \frac{\mathsf{T}' = \mathsf{T}[u\rho^{\mathsf{i}\to\mathsf{a}}]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\mathrm{actv}(u)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \rho^{\mathsf{i}}$$

$$\text{C-DEACT}\ \frac{\mathsf{T}' = (\mathsf{T}\backslash u) \mid u\rho^{\mathsf{i}}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\mathrm{deact}(u)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \rho^{\mathsf{a}}$$

$$\text{C-SEND}\ \frac{\mathsf{R}' = \mathsf{R} - \mathrm{peer}(\tilde{v} \cup \{u\}) \qquad \mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{!}] \qquad Com' = Com[u' \triangleleft u : M(\tilde{v})]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u'\mathbf{!}M(\tilde{v})} B'(\mathsf{P},\mathsf{R}',\mathsf{T}'), Com'}\ \triangle$$

$$\text{C-RECV}\ \frac{\mathsf{T}' = \mathsf{T}[u\rho/u\mathbf{?}] \qquad \mathsf{R}' = \mathsf{R} \cup \{\mathrm{refs}(\tilde{v}), u''\} - \{u' | (u \multimap u') \wedge \mathrm{peer}(u')\} \qquad Com' = Com[u\triangleright] \quad \mathsf{P}' = \mathsf{P}[u \multimap u''] \text{ if } \mathrm{peer}(u)}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u''\mathbf{?}M(\tilde{v})} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ \bigtriangledown$$

$\square \triangleq (u \multimap \bot) \ \wedge\ \mathsf{T}(u) = \mathbf{!}^{\mathsf{a},\mathsf{i}} \ \wedge\ v \in \mathsf{R} \ \wedge\ \big(\mathrm{peer}(v) \Rightarrow v \notin CoDom(\mathsf{P})\big)$

$\triangle \triangleq \mathsf{T}(u) = \mathbf{!}^{\mathsf{a}} \ \wedge\ (u \multimap u') \ \wedge\ u' \in Com \ \wedge\ \mathrm{refs}(\tilde{v}) \subseteq \mathsf{R} \ \wedge\ \mathrm{peer}(\tilde{v}) \cap CoDom(\mathsf{P}) = \varnothing$

$\bigtriangledown \triangleq \mathsf{T}(u) = \mathbf{?}^{\mathsf{a}} \ \wedge\ Com.u_{\triangleright} = u'' : M(\tilde{v})$

Table 3. *Rules for component semantics*

contract described by this language. The definitions of Section 3 are extended with the notion of contract. This notion is based on observers: the contract observes the actions of the component; if the action respects the contract, component and contract will evolve together.

A component has a set of contracts, one for each port. We use the notation:

$u{:}T, u{<:}T$ u has the behaviour described by the type T ($u{:}T$) or a subtype of T ($u{<:}T$)

$(B, \tilde{C})$ contract $\tilde{C}$ is associated to B. $\tilde{C}$ is a set of $(u : T)$, such that each reference (ports and partners) has an associated type. Addition of a reference is denoted $\tilde{C} \Leftarrow u{:}T$ (if $u{:}T'$ is already in $\tilde{C}$, there is no addition), and removal $\tilde{C} \backslash u$. Modification[4] is denoted $\tilde{C}[u{:}T'/T]$ when the type of u changes from T to T', and $\tilde{C}[u'{:}T'/u{:}T]$ when the reference $u{:}T$ is replaced by another reference $u' : T'$. We also write $u{:}T$ for $u{:}T \in \tilde{C}$, when there is no ambiguity (especially in the rules).

4.1. *Rules for Contract Satisfaction: Valid Transitions*

The rules (Tab. 4) are based on the ones of Sect. 3, whereby Com is abstracted from the state structure. Contracts are known locally: there is no global environment gathering them.

The predicate $\mathrm{Must}(\mathsf{T})$ states that any port u of the thread T which is typed **must!** is not suspended by a port v which is typed **may?**. This predicate is to be verified each time a port can change its state to a reception, meaning after a sending or a receiving. There is no need to verify this predicate when a port suspends to another one, because component semantics forbids the latter port to be receiving.

CREAT allows a port to be created. It adds the type of the port to $\tilde{C}$.
REMVPORT concerns removal of a port: its type is removed from $\tilde{C}$. No condition relates to the rule, but an active port in reception which can possibly receive a message cannot be removed. Indeed, rule DEACT prohibits an active port in reception to become idle, except in an exceptional case, where the corresponding port will no longer receive messages.
REMVREF removes a reference from R. It is allowed only if the reference is not typed **must?**: otherwise the component will have to interact with this reference, or send it.
BIND is very important, as it concerns binding of a reference to a port. Port and partner must have compatible types. This enables dynamic links that are valid.
DEACT authorises deactivation in the following cases: u is not in a receiving state ($\langle\!\langle u{:}T \not\equiv mod\ \mathbf{?}\ [\star]M_\Sigma \rangle\!\rangle$), or the type of partner of u is a subtype of **0** (then u has the type **0** or **may?**). Regarding as to the second point, the port v will never send messages: corollary 2 in section 5.3 ensures that the real type of port v is a subtype of the type of the *reference* v, hence the *port* v is a subtype of **0**, and is not in a sending state[5].
SEND verifies the sendings. When u sends message M_k, then its type and the one of its partner u' will evolve accordingly. Sent references must be of the right type, and, for the peer references, removed from $\tilde{C}$ ($\langle\!\langle \mathrm{peer}(\tilde{v}_k \cap \overline{\mathsf{P}}) \rangle\!\rangle$ gives peer references that are not local ports).
RECV verifies receptions of messages, when the port is bound to a partner (the type of this partner is in $\tilde{C}$: $u' : T' \in \tilde{C}$). Once reception is made, the type of the port and the one of its

4. Concerning servers: the contract keeps the initial type of the server. If $(B, \tilde{C})$ uses reference $g^* : T$, once the first message is sent, $\tilde{C}$ will contain both $g^* : T$ and $g : T'$, with T' the next type of the server.
5. The demonstration of the corollary given in section 5.3 is done by induction on the rules; the reasoning we gave may appear circular, but the corollary and this proof can be shown without rule DEACT.

For the sake of readability, we abbreviate:
$M_\Sigma \triangleq \mathbf{[}\Sigma_k M_k(\tilde{U}_k)\mathbf{;}\ T_k\mathbf{]}, M'_\Sigma \triangleq \mathbf{[}\Sigma_k M_k(\tilde{U}'_k)\mathbf{;}\ T'_k\mathbf{]} \quad m_k \triangleq M_k(\tilde{v}_k)$

$$\text{CREAT}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{creat}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\mathrm{creat}(u)} (B'(\mathsf{P},\mathsf{R}',\mathsf{T}'),\tilde{C} \Leftarrow u{:}T)}$$

$$\text{REMVPORT}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\mathrm{remvport}(u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}$$

$$\text{REMVREF}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\mathrm{remvref}(u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}\ (T \not\equiv \textbf{must?}\ M_\Sigma)$$

$$\text{BIND}\ \frac{u{:}T \qquad v{:}S \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{bind}(u \multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\mathrm{bind}(u \multimap v)} (B'(\mathsf{P}',\mathsf{R},\mathsf{T}),\tilde{C})}\ Comp(T,S)$$

$$\text{DEACT}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{deact}(u)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\mathrm{deact}(u)} (B'(\mathsf{P},\mathsf{R},\mathsf{T}'),\tilde{C})}\ \left(\begin{array}{c} u{:}T \not\equiv mod\ \textbf{?}\ [\textbf{*}]M_\Sigma \\ \vee\big((u \multimap v)\ \wedge\ v <: \mathbf{0}\big) \end{array}\right)$$

$$\text{SEND}\ \frac{u{:}T \equiv mod\ \textbf{!}\ M_\Sigma \qquad u'{:}T' \equiv mod'\ \textbf{?}\ [\textbf{*}]M'_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\textbf{!}m_k} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u'\textbf{!}m_k} (B'(\mathsf{P},\mathsf{R}',\mathsf{T}'),\tilde{C}[u{:}T_k/T, u'{:}T'_k/T']\backslash\{\mathrm{peer}(\tilde{v}_k \cap \overline{\mathsf{P}})\}}\ \blacktriangleleft$$

$$\text{RECV}\ \frac{\begin{array}{c} u{:}T \equiv mod\ \textbf{?}\ M_\Sigma \\ u'{:}T' \equiv mod'\ \textbf{!}\ M'_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u''\textbf{?}m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}') \end{array}}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u''\textbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}[u{:}T_k/T, u''{:}T'_k/u'{:}T'] \Leftarrow \tilde{v}{:}\tilde{U}'_k)}\ \blacktriangle \wedge (u \multimap u') \in \mathsf{P}$$

$$\text{RECV-UN}\ \frac{u{:}T \equiv mod\ \textbf{?}\ M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\textbf{?}m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u'\textbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}[u{:}T_k/T] \Leftarrow u'{:}{T_k}^{\mathcal{D}},\tilde{v}{:}\tilde{U}_k)}\ \blacktriangle \wedge (u \multimap \bot) \in \mathsf{P}$$

$$\text{RECV*}\ \frac{u{:}T \equiv mod\ \textbf{?*}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:w\textbf{?}m_k,\mathrm{creat}(u'),\mathrm{bind}(u' \multimap w),\mathrm{actv}(u')} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u'/u^\star:w\textbf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C} \Leftarrow u'{:}T_k, w{:}{T_k}^{\mathcal{D}},\tilde{v}{:}\tilde{U}_k)}\ \blacktriangle$$

$$\text{OTHER}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\alpha} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\alpha} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C})}\ \alpha \in \{\mathrm{actv}\}$$

$$\blacktriangleleft \triangleq \tilde{v}_k <: \tilde{U}_k \wedge \mathrm{Must}(\mathsf{T}') \qquad \blacktriangle \triangleq len(\tilde{v}) = len(\tilde{U}_k) \wedge \mathrm{Must}(\mathsf{T}')$$

$$\mathrm{Must}(\mathsf{T}') \triangleq \forall u \in \mathsf{T}', (u{:}\textbf{must!}M_\Sigma) \Rightarrow \forall v, u \rightarrowtail^* v : \neg(v{:}\textbf{may?}M_\Sigma)$$

Table 4. *Rules for contract satisfaction (valid transition)*

partner evolve accordingly. Concerning the partner, modification in $\tilde{C}$ associates the type of the old partner (u', whose type T' becomes T'_k after reception) to the new partner (u'' which is then typed T'_k). Corallary 2 ensures that the true type of u'' is a subtype of T'_k.
RECV-UN deals with receptions when the port is not bound to a partner. Thus the type of the partner is unknown. We choose to associate to this partner the greater possible supertype, which is the dual type of the port receiving the message (prop. 3.4). Received references are stored with type $\tilde{U}_k$, which is necessarily a supertype of the real type of the references.
RECV* checks the correct operations of a server: the component must create a port at once, which will answer the request (sequence «$u : w\mathbf{?}m_k, \mathrm{creat}(u'), \mathrm{bind}(u' \multimap w), \mathrm{actv}(u')$»). Just like rule RECV-UN, the partner is given the dual type of that of the server port.
OTHER gives all other valid transitions, where the set $\tilde{C}$ is not modified.

We do not check the type of received arguments, because the message was sent according to the type of the sender; as the sender has to be compatible with the receiver, we are sure the arguments are well-typed. We aim also at an optimistic verification, where the environment has to respect the interface semantics: received messages not allowed by the type are not taken into account. Interactions between ports of the same component is also a bit tricky. Given u and v such ports, the component can behave such that u sends messages to v, knowing v is in the same component. Consequences of such a behaviour enforces the environment not to interact with port v. Just like the environment must respect the interface type it simulates, it must also respect privacy of peer references (i.e. it does not know reference v in this case).

4.2. *Rules for Contract Satisfaction: Invalid Transitions*

Rules of Tab. 5 give the error cases, where the component does not respect its contract.

REMVREF-ERR raises an error when a reference typed **must?** is removed from R.
BIND-ERR states that port and reference cannot be bound if their types are not compatible.
DEACT-ERR forbids a port in a receiving state ($u : mod$ **?** ...) to become idle if it has no partner, or if this partner can possible send messages ($v \not<: \mathbf{0}$). This ensures corollary 1 (sect.5).
SEND-ERR concerns errors while sending messages. Two cases arise: the message is not in the list of authorised sendings, and when the type is asking for a reception, or is **0**.
RECV-ERR concerns reception of messages not performed.
RECV*-ERR deals with server ports in the component, when a port is not immediately created upon the reception of the request.
MUST-ERR is used when predicate $\mathrm{Must}(\mathsf{T}')$ is false (meaning: a port typed **must!** is suspended by a port typed **may?**; consequence is that the first port may not honour its contract).

4.3. *Component Honouring a Contract*

A component honouring a contract, noted $B(\mathsf{P}, \mathsf{R}, \mathsf{T}) \models \tilde{C}$, is such that the reduction process will never lead to *Error*:

$$B(\mathsf{P}, \mathsf{R}, \mathsf{T}) \models \tilde{C} \quad \text{iff} \quad \forall B', \tilde{C}' \text{ such that } (B, \tilde{C}) \rightarrow^* (B', \tilde{C}') : \quad (B', \tilde{C}') \nrightarrow \mathit{Error}$$

5. Sound Assembly of Components: Definition and Properties

So far, we defined compatibilities between a component and its interface types, and between interfaces. In this section, we investigate properties on an assembly of components,

For the sake of readability, we abbreviate:
$M_\Sigma \triangleq \mathbf{[}\Sigma_k M_k(\tilde{U}_k)\mathbf{;}\ T_k\mathbf{]}, M'_\Sigma \triangleq \mathbf{[}\Sigma_k M_k(\tilde{U}'_k)\mathbf{;}\ T'_k\mathbf{]} \quad m_k \triangleq M_k(\tilde{v}_k)$

$$\text{REMVREF-ERR}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ T \equiv \textbf{must?}M_\Sigma$$

$$\text{BIND-ERR}\ \frac{u{:}T \qquad v{:}S \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{bind}(u \multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \neg \mathit{Comp}(T,S)$$

$$\text{DEACT-ERR}\ \frac{u{:}T \equiv \mathit{mod}\ \textbf{?}\ [\textbf{*}]M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{deact}(u)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \begin{pmatrix} (u \multimap \bot) \\ \vee\big((u \multimap v) \wedge v \not<: \mathbf{0}\big) \end{pmatrix}$$

$$\text{SEND-ERR}\ \frac{u{:}T \equiv \mathit{mod}\ \rho\ [\textbf{*}]M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\textbf{!}m'} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \neg m'{:}M_\Sigma \vee \rho = \textbf{?},\mathbf{0}$$

$$\text{RECV-ERR}\ \frac{u{:}T \equiv \mathit{mod}\ \textbf{?}[\textbf{*}]M_\Sigma \qquad \forall k, B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'\textbf{?}m_k} \!\!\!\!/\ \ B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}$$

$$\text{RECV*-ERR}\ \frac{u{:}T \equiv \mathit{mod}\ \textbf{?*}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:w\textbf{?}m_k,\text{creat}(u'),\text{bind}(u' \multimap w),\text{actv}(u')} \!\!\!\!/\ \ B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}$$

$$\text{MUST-ERR}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \rightarrow B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \neg\text{Must}(\mathsf{T}')$$

$$\neg m'{:}M_\Sigma \triangleq m' = M'(\tilde{v}') \wedge \forall k, M' \neq M_k \vee \neg\tilde{v}_k{:}\tilde{U}'_k$$

Table 5. *Rules for contract satisfaction (invalid transitions)*

and prove safety properties (no error occurs, and no deadlock between ports will occur), and liveness properties (all messages sent are eventually consumed).

For the sake of readability, and to abstract from component structure and its contract, we denote $u : T \in \mathsf{R}$ to indicate that, in the contracted component $B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}$: $u \in \mathsf{R}$ and $u : T \in \tilde{C}$. Also, we write $(u : T \multimap v : T')$ to state that, in $B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}$, we have $(u \multimap v) \in \mathsf{P}$ and $u{:}T, v{:}T' \in \tilde{C}$.

5.1. *Assembly of Components*

We defi ne an assembly of components as a confi guration of components with their contract, and ready to interact via a communication medium. An assembly, in its initial confi guration, encompasses both client/server and peer-to-peer bindings. It has the following properties, where the last two ensure that a peer reference is private [NNS99]:

– the confi guration is reference-closed: references designate ports in the confi guration;
– all the ports are active on independent threads;
– peer ports in reception have a unique partner in the confi guration;
– a component knows no peer references, except those of his own ports and their partners.

Definition 1 (Assembly of Components)
$\mathcal{A} = \{(B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1), \tilde{C}_1), \ldots, (B_n(\mathsf{P}_n, \mathsf{R}_n, \mathsf{T}_n), \tilde{C}_n), Com\}$

$$\textit{with the 4 properties:} \begin{cases} \forall i, u : & u \in \mathsf{R}_i \Rightarrow \exists j \textit{ such that } u \in \mathsf{P}_j \\ \forall u \in \cup \mathsf{P}_i : & \mathsf{T}(u) = \rho^{\mathsf{a}} \\ \forall u \in \mathrm{peer}(\cup \mathsf{P}_i), u\mathsf{?}^{\mathsf{a}} : & \exists! v, j \textit{ such that } (v \multimap u) \in \mathsf{P}_j \\ \forall i : & u \in \mathsf{R}_i \Rightarrow u \in \mathsf{P}_i \ \vee \ \exists v : (v \multimap u) \in \mathsf{P}_i \end{cases}$$

Then, a sound assembly is an assembly where each component satisfi es its interface contracts, and linked ports have their interfaces mutually compatible:

Definition 2 (Sound Assembly)
$\mathcal{A}$ *is sound (denoted* $\models \mathcal{A}$*) iff*
$\forall i : \big(B_i \models \tilde{C}_i\big) \ \wedge \ \big((u{:}T_u \multimap v{:}T_v) \in \mathsf{P}_i \Rightarrow (Comp(T_u, T_v) \ \wedge \ \exists!\, j, T' \textit{ such that } v <: T' \in \mathsf{P}_j)\big)$

5.2. *Compatibility and Message Consumption Properties*

Compatibility relation is very important in our study. The following property ensures all references bound to port are such that their respective type are compatible.

Property 4 (Safeguarding of Compatibility)
If $\models \mathcal{A}$, *then* $\forall \mathcal{C}, \mathcal{A} \rightarrow^* \mathcal{C}$, *we have:* $\forall u, v \in \mathcal{C}, (u{:}T_u \multimap v{:}T_v) \Rightarrow Comp(T_u, T_v)$.

Proof. By structural induction, and from the defi nition of the compatibility relation (interfaces evolve towards compatible types). □

The next property, P_{sr}, of a sound assembly states simply that soundness is maintained throughout the evolution: a confi guration of component never leads to*Error*:

Theorem 1 (Subject Reduction)
If $\mathcal{A}$ *is sound, then* $\mathcal{A} \vDash P_{\mathrm{sr}}$, *with* $P_{\mathrm{sr}} \triangleq \forall \mathcal{C} : \mathcal{A} \rightarrow^* \mathcal{C}, \mathcal{C} \nrightarrow Error$.

Proof. By structural induction on the transition rules. The property is satisfi ed by observing that the only way a confi guration can lead to*Error* is by violating compatibility rules. □

Corollary 1 (Message Consumption)
If $\mathcal{A}$ *is sound, then* $\mathcal{A} \vDash P_{\mathrm{mc}}$,
with: $P_{\mathrm{mc}} \triangleq \forall u, v, i, M : (u \multimap v) \in \mathsf{P}_i, (\mathcal{C} \xrightarrow{u:v!M} \mathcal{C}') \Rightarrow \exists \mathcal{C}'', \mathcal{C}''' : \quad \mathcal{C}' \rightarrow^* \mathcal{C}'' \xrightarrow{v:u?M} \mathcal{C}'''$

Proof. This corollary is a consequence of theorem 1, the use of FIFO queues and the constraint that a port in reception is active. □

The corollary ensures that in the future, a transition will be able to consume the message. However, since the rules compete with each other, we have to assume fairness in this competition. Hence, the corollary ensures the consumption of the message in the future, but modulo fairness.

5.3. *Type of a Reference in the Configuration*

This section presents an interesting property that relates the type of a port and the type of the reference pointing at it. The following lemma gives such a relation.

Lemma 2
Given $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ *and* $B(\mathsf{P}, \mathsf{R}, \mathsf{T})$*, with* $u \notin \mathsf{R}_u$*,* $u{:}T_{\text{part}} \in \mathsf{R}$*, and* $Com.u = \varnothing$*. Then:*

$$(u{:}T \multimap v{:}T') \in \mathsf{P}_u \Rightarrow {T'}^{\mathcal{D}} \preceq T_{\text{part}}$$

$$(u{:}T \multimap \bot) \in \mathsf{P}_u \Rightarrow T \preceq T_{\text{part}}$$

The lemma applies only when reference u and the corresponding port are in different components ($u \notin \mathsf{R}_u$); type comparison is possible only if queues are empty, and shows that:

– if port u is bound to v: the type of reference u is a supertype of the dual of the type of v;
– if port u is unbound: the type T_{part} of reference u is a supertype of the type of u.

However, we will rather use the corollary which rises from this lemma, namely that the type T_{part} of reference u is supertype of the type of port u.

Corollary 2 (relations between the type of a reference and the type of the associated port)
$\forall u$ *such that* $Com.u = \varnothing$*:*

$$u{:}T_{\text{part}} \in \mathsf{R} \ \wedge \ u{:}T \in \mathsf{P}_u \Rightarrow T \preceq T_{\text{part}}; \qquad \text{if } u \in \mathsf{R}_u \text{ we have } T = T_{\text{part}}.$$

Proof. Straightforward from lemma 2 and property 3.5 ($Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$). The case $u \in \mathsf{R}_u$ has two possibilities: port u has not yet been sent outside the component ($T = T_{\text{part}}$ is then obvious), or port u has been sent, and was received later on by its own component. At the time of the reception of the reference, $u : T'$, is saved with $\tilde{C} \Leftarrow u : T'$, which does not modify the type of u in $\tilde{C}$; equality $T = T_{\text{part}}$ follows. □

Proof (lemma 2, sketch).
By induction. As queues have to be empty, we consider that messages are consumed immediately. Note that the lemma is verifi ed during the assembly, from property 3.5.

Demonstrations for rules CREAT, REMVPORT, REMVREF, DEACT and OTHER are straightforward. As for BIND, conclusion is quite obvious; concerning the bound partner: hypotheses are not changed while applying the rule. Concerning the port: the latter is sending; as we had, before the binding, $(u \multimap \bot)$, necessarily $u \in \mathsf{R}_u$ because a port which is sending cannot be sent. The lemma is then not concerned by this case.

Concerning rules SEND, RECV, RECV* and RECV-UN: we suppose a reception happens right after a sending. We have the following cases:

references $\tilde{v}_k$ *in the arguments of the message*
Reasoning is straightforward, by looking at compatibility and subtype relations: while passed through messages, references are associated to a supertype of their current type.

unknown partner at the time of the reception (rule SEND followed by RECV-UN or RECV*)
We use the dual property (Sect. 2.5). Note that, in the rule, the dual is applied only to peer references. Before applying the rules, we had (with u sending and v receiving):

$$(u{:}T \multimap v{:}T'_{\text{part}}) \qquad (v{:}T' \multimap \bot) \text{ with, by induction } T' \preceq T'_{\text{part}}$$

Let's write T_k, T'_{part_k} and T'_k the types resulting from the sending or receiving of message M_k. When u applies SEND, and v applies one of the RECV rules, we have:

$$(u{:}T_k \multimap v{:}T'_{\text{part}_K}) \qquad (v{:}T'_k \multimap u{:}{T'_k}^{\mathcal{D}}) \text{ with } T'_k \preceq T'_{\text{part}_k}$$

Concerning u, we use the induction hypothesis $T'_k \preceq T'_{\text{part}_k}$ and property 3.5 to prove $(T'_{\text{part}_k})^{\mathcal{D}} \preceq {T'_k}^{\mathcal{D}}$. Concerning v, $({T'_k}^{\mathcal{D}})^{\mathcal{D}} \preceq T'_{\text{part}_k}$ is straightforward (${T'_k}^{\mathcal{D}^{\mathcal{D}}} = T'_k$).

known partner at the time of the reception (rule SEND followed by RECV)
With the same reasoning (with u sending, v receiving, w equal or different from u):

$$(u{:}T \multimap v{:}T'_{\text{part}}) \qquad (v{:}T' \multimap w{:}T'') \text{ with } {T''}^{\mathcal{D}} \preceq T'_{\text{part}}$$

When u applies SEND and v RECV, we have:

$$(u{:}T_k \multimap v{:}T'_{\text{part}_k}) \qquad (v{:}T'_k \multimap u{:}T''_k) \text{ with } {T''_k}^{\mathcal{D}} \preceq T'_{\text{part}_k}$$

Concerning v, ${T''_k}^{\mathcal{D}} \preceq T'_{\text{part}_k}$ is true. We easily deduce, for the port u: $(T'_{\text{part}_k})^{\mathcal{D}} \preceq T''_k$ from $({T''_k}^{\mathcal{D}})^{\mathcal{D}} = T''_k$ and property 3.5. □

5.4. *External Deadlock Freeness*

External deadlock represents the situation where a set of ports are inter-blocked because of a dependency cycle. The simplest form of external deadlock is written:

$$(u \multimap u') \wedge (v \multimap v') \wedge (u\mathbf{!}^{\mathsf{s}} \rightarrowtail v\mathbf{?}^{\mathsf{a}}) \wedge (v'\mathbf{!}^{\mathsf{s}} \rightarrowtail u'\mathbf{?}^{\mathsf{a}})$$

u sending is blocked by v which is waiting for v' to send which, in turn, is blocked by u', which is waiting for u to send.

The general case can be formalised, on the principle of *Wait-For-Graph*, as the existence of cycles in a dependency graph. We introduce a new dependency relation between the ports, named external dependency, denoted $\cdot \dashrightarrow \cdot$. This dependency relates the communications between distant ports. For example, $u\mathbf{?}^{\mathsf{a}} \dashrightarrow v\mathbf{!}^{\mathsf{s}}$ means u depends on v, more precisely u waits for v to send.

Definition 3 (External dependency $\dashrightarrow$)
$u \dashrightarrow v$ *iff* $u\mathbf{?}^{\mathsf{a}} \wedge \big((v\mathbf{!}^{\sigma} \wedge (v \multimap u)) \vee (v\mathbf{?}^{\mathsf{a}} \wedge (u \multimap v) \wedge u : M(\ldots) \in Com.v)\big)$

Remarks: if u and v are both typed **may?**, all queues are empty and there is no dependency. In the other cases, the fact that v can be in reception ($u\mathbf{?}^{\mathsf{a}} \dashrightarrow v\mathbf{?}^{\mathsf{a}}$) is a particular case which is due to asynchronous messages: this case shows that u has just sent a message M and waits a response from v, whereas v has not yet consumed M.

A deadlock is then formalised as follow:

Definition 4 (Ext_deadlock)
$\text{Ext\_deadlock}(\mathcal{C}) \triangleq \exists (u_i)_{1..n} \in \mathcal{C}$ *such that* $\forall k < n : u_i \mathcal{S} u_{i+1} \wedge u_n \mathcal{S} u_1$
with $u\mathcal{S}v = true$ *when* u *is suspended to* v: $u\mathcal{S}v \triangleq u \rightarrowtail v \vee u \dashrightarrow v$

Note that idle or active ports in a sending state are not concerned by a deadlock.

Theorem 2 (External deadlock freeness)
If $\mathcal{A}$ is sound, then $\mathcal{A} \vDash P_{\text{edf}}$ *with* $P_{\text{edf}} \triangleq \forall \mathcal{C}, \mathcal{A} \rightarrow^* \mathcal{C} \Rightarrow \neg\text{Ext\_deadlock}(\mathcal{C})$

The deadlock freeness problem has received attention recently. A work on this issue which is very close to ours is the one by Naoki Kobayashi [Kob02], where the author does have **may** and **must** actions (in terms of capabilities and obligations), but communications are synchronous, and constraints between compatible types are too restrictive for our study.

Proof of theorem 2 is tedious. Even if interfaces are mutually compatible, it is not straightforward that a deadlock will not arise between components (ports in a component may be suspended by other ports, which leads to potential dependencies between threads added to dependencies between ports). The proof needs two lemma, presented hereafter. The first one indicates where a peer reference is.

Lemma 3
Given a peer reference v, one and only one of the following alternatives hold:

1) if $\exists B(\mathsf{P},\mathsf{R},\mathsf{T})$ such that $v \in \mathsf{R}$, then B is unique;
2) $\exists!(u, M)$ such that $M(\ldots, v, \ldots) \in Com.u$; moreover $\nexists B(\mathsf{P}, \mathsf{R}, \mathsf{T})$ such that $v \in \mathsf{R}$;
3) $\exists M(\ldots)$ such that $v : M(\ldots) \in Com.u \ \wedge \ (u \not\multimap v)$; moreover $\nexists B(\mathsf{P}, \mathsf{R}, \mathsf{T})$: $v \in \mathsf{R}$;

Proof. By structural induction. We present here the non obvious cases.

• *Rule SEND*: u sends message $M_k(\tilde{v})$ towards u'. We will use the notation $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ for a component for each $u \in \mathsf{P}_u$. We consider three kinds of references:

a peer reference $v \in \tilde{v}$ is in an argument of a message
By induction, B_u is the unique component knowing v ($v \in \mathsf{R}_u$); when v is sent, it is removed from R_u. Hence, after applying SEND, v is known of no component (point 1. of the lemma is false), and there is a unique message in Com such that v is in the arguments of the message (point 2. of the lemma is true). Point 3. of the lemma is also false.

reference u' which receives the message
Lemma remains true for u': only the first point of the lemma is true ($u' \in \mathsf{R}_u$).

reference u that sent the message
Considering the case where u' is bound to u, we have $u \in \mathsf{R}_{u'}$, and points 2. and 3. of the lemma are false. Considering the case where $(u' \not\multimap u)$, two possibilities arise:

u is sending for the first time towards u'. First, let's look at the binding of u' to port u. It may be due either to a reception of a message or to the BIND rule. The first case is impossible because $(u' \not\multimap u)$, so u' could not have send the message. Hence, $(u \multimap u')$ is due to the BIND rule, and no reception has occurred since. As u is sending for the first time, u has made no action between BIND and SEND. Thus, we know that since u has been created, it was always in a sending state. As this is its first sending, we have, before applying SEND: $u \in \mathsf{R}_u$, thus only point 1. of the lemma is true. After the sending, only point 3. of the lemma is true.

u has already sent a message towards u'. Point 3. of the lemma was true, which remains true after the sending (note that the first message that u sent is not yet consumed by u', because $(u' \not\multimap u)$).

• *Rules RECV, RECV*, RECV-UN* (sketch): reasoning is the same as for rule SEND: we give here only a sketch. Considering message $u' : M(\tilde{v})$ received by u, we have three cases:

references as argument of the received message
Concerning those references, point 3. of the lemma is true before the reception; only point 1. is true after the reception.

the reference u' that sent the message
In the case where $(u \multimap u')$ before the reception, then there is no modification (only point 1. of the lemma is true). In the case where u' is a new partner for u, then point 3. of the lemma was true before the reception, and only point 1. is true after.

reference u that receives the message
There are no changes during the application of the rule. □

The second lemma concerns external dependency relations between peer ports. It shows that this relation is unique. It is a consequence of lemma 3.

Lemma 4
If u and v are peer ports such that $u \dashrightarrow v$, then u and v are unique.

Proof. Given u and v such that $u \dashrightarrow v$. By definition:
$$u\mathbf{?}^{\mathsf{a}} \wedge \big((v\mathbf{!}^{\sigma} \wedge (v \multimap u)) \vee (v\mathbf{?}^{\mathsf{a}} \wedge (u \multimap v) \wedge u : M(\ldots) \in Com.v)\big)$$
We have two cases (we denote $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ the component B_u such that $u \in \mathsf{P}_u$):

v is sending: $v\mathbf{!}^{\sigma} \wedge (v \multimap u)$.
As v can have only one partner, there is only one u such that $u \dashrightarrow v$. Concerning the unicity of v: as we have $(v \multimap u)$, necessarily $u \in \mathsf{R}_v$ (by looking at how a binding is made – through BIND and RECV rules – and also because a bounded reference cannot be sent). Suppose there exists $B_w(\mathsf{P}_w, \mathsf{R}_w, \mathsf{T}_w)$ whereby $w \neq v$ and $(w \multimap u)$; then we have $u \in \mathsf{R}_w$. Lemma 3 ensures $B_w = B_v$. As it is not possible to have, in the same component, a reference bound to two different ports, we have $w = v$ and v is unique.

v is receiving: $v\mathbf{?}^{\mathsf{a}} \wedge (u \multimap v) \wedge u : M(\ldots) \in Com.v$
The same reasoning applies (suppose there exists $w \neq u$ having the same properties than u, then $w = u$ necessarily). □

The latter lemma is used in the demonstration of theorem 2:

Proof (External Deadlock Freeness).
By structural induction on the contract rules. CREAT, REMVPORT, REMVREF and DEACT do not create any dependency relation, thus are not concerned. As for BIND, a dependency relation is created, but as only active or idle ports in a sending state can apply this rule, and as those ports are not concerned by predicate $\mathrm{Ext\_deadlock}(\mathcal{C})$, no deadlock is created.

• *Rule SEND*: We distinguish between the next action ρ of port u:

$\rho = \mathbf{!}$ dependency relations remain unchanged;

$\rho = \mathbf{0}$ dependency between u and its partner disappears;

$\rho = \mathbf{?}$ this is the particular case $u\mathbf{?}^{\mathsf{a}} \dashrightarrow v\mathbf{?}^{\mathsf{a}}$, with v the partner of u: $(u \multimap v)$.
Reductio ad absurdum: suppose this dependency created a deadlock:

$$\exists (u_i)_{1..n} \in \mathcal{C} \text{ such that } \forall k : u_i \mathcal{S} u_{i+1} \text{ with } u, v \in (u_i)$$

Let's chose (u_i) such that $u_1 = u$ and $u_2 = v$ (as we have $u\mathcal{S}v$) and let's show that if $v\mathcal{S}w$ then $w = u$. As v is active, the only dependency between u and v is an external one, and by lemma 4 we have $w = u$. Hence we have, simultaneously:

$$u \dashrightarrow v \text{ that is to say } v\mathbf{?}^{\mathsf{a}} \ \wedge \ (u \multimap v) \ \wedge \ u : M(\ldots) \in Com.v$$
$$v \dashrightarrow u \text{ that is to say } u\mathbf{?}^{\mathsf{a}} \ \wedge \ (v \multimap u) \ \wedge \ v : M'(\ldots) \in Com.u$$

In particular, we have $v : M'(\ldots)$ in $Com.u$, which is contradictory since u has just sent a message, and types of u and v are compatible (so u would have made more receptions that the number of sendings from v).

• *Rules RECV, RECV-UN* (sketch): as for SEND, we have three cases, depending on ρ:

$\rho = \mathbf{?}$ if an external dependency arises, it is identical to the one before the reception.

$\rho = \mathbf{0}$ dependency between u and its partner disappears;

$\rho = \mathbf{!}$ then we have $\mathsf{T}(u) = \mathbf{!}^{\mathsf{a}}$. u is not concerned by $\mathrm{Ext\_deadlock}(\mathcal{C}')$, hence the predicate is false (by induction).

• *Rule RECV**: Upon the reception of the message, a new port u' is created, with its own thread of execution. Hence, there is a new external dependency, but no deadlock arises.

• *Rule OTHER*: This rule concerns other transitions of the component, namely C-ACTV and C-ACTV2. The rule C-ACTV adds dependency $u\mathbf{!}^{\mathsf{s}} \rightarrowtail v\mathbf{!}^{\mathsf{a}}$; the port v is not concerned by predicate $\mathrm{Ext\_deadlock}(\mathcal{C}')$. Conclusion for C-ACTV2 is straightforward, as a port is activated on its own thread of execution. □

5.5. *Liveness Properties under Assumptions*

The assembly of components may have still a livelock problem: a port can be forever suspended because of a divergence of some internal computation or an endless dialogue between two ports. Thus it is not possible to prove a liveness property that states "each port reaching a **must?** (or **must!**) state will eventually receive (or send) a message":

$$P_{\text{must}} \triangleq \forall\, \mathcal{C}, u, i : \ \ \mathcal{A} \rightarrow^* \mathcal{C}, (u{:}\mathbf{must}\rho M_\Sigma) \in \tilde{C}_i \text{ with } \rho \in \{\mathbf{?}, \mathbf{!}\} \Rightarrow$$
$$\exists \mathcal{C}', \mathcal{C}'', v \text{ such that } \mathcal{C} \rightarrow^* \mathcal{C}' \xrightarrow{u:v\rho M_k} \mathcal{C}''$$

However, we believe this liveness property is verifi ed with the assumptions:

– a computation in a component always ends;
– a suspended port which becomes active must send its message before suspending again;
– a port which has a loop behavior will become idle in the future.

Anyhow, these properties can only be verifi ed at a lower level of abstraction, that is, only when the concrete behaviour of the components (e.g. its source code) is known.

6. Bank Account Example

To illustrate our framework, we use yet another bank account example: a client has to authenticate himself to the bank, so he can perform deposit and withdrawal operations on his bank account. We propose an implementation with three components: the `Client`, the `Bank` and the `Account`.

Several possible implementations exist; for example the `Client` receives a reference to his `Account` so he can interact with it. We propose a more elegant solution, described by figures 2 and 3: the`Client` will use the same port for the authentication and the operations on his account. The `Bank` identifies the client (figure 2: the client was correctly authenticated), and sends to the `Account` the reference of the port of the `Client`. `Account` will use this reference to inform the `Client` that on the one hand access is granted, and on the other hand the dialogue continues with `Account`. Figure 3 shows that if the user is not correctly identified, the `Bank` will return the `refused` message. The salient issue of this protocol is that the couple (`Bank`, `Account`) can be replaced by only one component, *without any change of* `Client`.

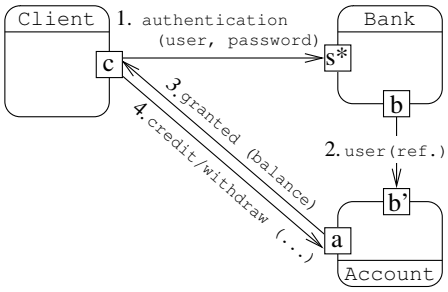


Figure 2. *Example: access to bank account granted*

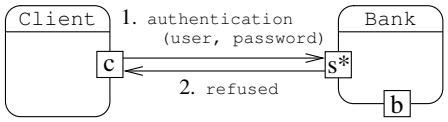


Figure 3. *Example: access to bank account refused (component* `Account` *omitted)*

6.1. *Interface Types*

The `Bank` component has two ports: s^* is a server port for authentications, and b is used to delegate operations to the `Account` component. The type of s^* is described hereafter:

bank_access = **may ? *[** *authentication* (string, string)**; must ! [** *granted* (real)**;** operations **+** *refused***; 0]]**

This server type can receive the *authentication* message (with user and password); after receiving this message, the port must either send *granted* with the balance and become *operations*, or send *refused* and stop. Type *operations* is described further.

In the case where the `Client` is correctly authenticated, the `Bank` delegates the sending of the *granted* message to the `Account`. This is done via port b, which is specified by the type *send_ref*. This type must send a reference which is typed *granted_user*: this latter type must receive only the *granted* message, and then become *client_operations*, the dual of *operations*.

send_ref = **must ! [** *user* (granted_user)**;** send_ref **]**
granted_user = **must ? [** *granted* (real) **;** client_operations **]**
client_operations = operations$^{\mathcal{D}}$

The `Account` component has also two ports: b' and a. The former is used to receive the reference of the client (type *received_ref*). The latter port, a, will interact with the `Client`. His type *access_granted* is written hereafter.

receive_ref = **may ? [** *user* (granted_user)**;** receive_ref **]**
access_granted = **must ! [** *granted* (real) **;** operations **]**

The type *operations*, which specifi es deposits and withdrawals, is described in two times:

operations = **may ? [** *deposit* (real)**;** **must ! [** *balance* (real)**;** operations **]**
+ *withdraw* (real)**; must ! [** *balance* (real)**;** operations
+ *neg_balance* (real)**;** negbal_operations **]]**
negbal_operations =
must ? [*deposit* (real)**;** **must ! [** *balance* (real)**;** operations
+ *neg_balance* (real)**;** negbal_operations **]**
+ *withdraw* (real)**; must ! [** *neg_balance* (real)**;** negbal_operations **]]**

A port typed *operations* can receive two messages: *deposit* and *withdraw*. After receiving one of the two messages, the port must send back the balance of the bank account: message *balance* is sent when balance is positive, and the type becomes *operations* again. Message *neg_balance* is sent when the user is debtor, and the type becomes *negbal_operations*.

The difference between the two types *operations* and *negbal_operations* is the modality: *operations* **may** receive messages, whereas *negbal_operations* **must** receive messages. Hence, as long as the client is debtor, he must perform some operation on his bank account.

Finally, the `Client` has one port, c, whose type can be written as follows. With this type, the `Client` will try to perform one operation. However, if this operation results in a negative balance, then other operations have to be made.

client = **must ! [** *authentication* (string, string)**;**
must ? [*granted* (real)**;** client_simple_operation
+ *refused***; 0]]**
client_simple_operation =
must ! [*deposit* (real)**;** **must ? [** *balance* (real)**; 0**
+ *neg_balance* (real)**;** client_simple_operation **]**
+ *withdraw* (real)**; must ? [** *balance* (real)**; 0**
+ *neg_balance* (real)**;** client_simple_operation **]]**

We have the following relations between the types:

Compatibilities to be checked during assembly	$Comp$(client, bank_access) $Comp$(send_ref, receive_ref)
Other compatibilities	$Comp$(granted_user, access_granted) $Comp$(client_waitauth, access_granted) $Comp$(client_simple_operation, operations) $Comp$(client_simple_operation, negbal_operations)
Other relations	client_waitauth $\preceq$ granted_user client_simple_operation $\preceq$ client_operations granted_user = access_granted$^{\mathcal{D}}$

with: client_waitauth = **must ? [** *granted* (real)**;** client_simple_operation **+** *refused***; 0]**

6.2. *Component Specification and Contract Verification*

For lack of space, we present only the most interesting component, that is `Bank`. Figure 4 shows in one shot the specification and the contract verification of the `Bank` component.

The specification is shown as simple state-charts (each bold rectangle representing a state $B(\mathsf{P}, \mathsf{R}, \mathsf{T})$. The greyed parts represent the changes compared to the previous state); transitions are labelled with actions as named in the rules. Reception of message *authenticate(...)*, in the upper part of the diagram, makes the component create a new port, s', to answer the client, which corresponds to the three states *Create Child*. This sequence terminates with "$\mathrm{actv}(s')$", which has two effects:

– go back to the state *Access* to answer new requests;

– create a new thread of execution. This thread will identify the user (state *Check Access*). If the user is not correctly identified (left branch), message *refused* is sent, and ports removed. If the user is correctly identified (right branch), port b is activated and sends message *user(c)* towards port b' of the `Account` component; note that c cannot be sent if bound to s', so we first have to deactivate and remove port s'. Also, as c is a peer reference, it has to be removed from R when sent to b'.

Contracts associated to each state are shown with parallelograms. Greyed parts show the verifications that are made. Once message *authenticate(...)* is received, the component performs the correct sequence of actions. The contract associated with the *Check Access* state contains both the type of the server port s^* and the type of the created port s'. Note also an application of corollary 2: the type of reference c (*client_operations*) is the dual of s', and is a super-type of the type of port c (*client_simple_operation*).

7. Conclusion & Future Work

We have presented a concept of behavioural contracts that we applied on a component model featuring multiple threads, reference passing, peer-to-peer and client/server communication patterns. Our contracts serve for the early verification of compatibility between components, in order to guarantee safety and liveness properties. Compatibility is formally described in this framework, as a composition of internal compliance of components to their interfaces, and conformance between interfaces.

In the context of component based design, the verification that a component is honouring a contract given by its interfaces is in charge of the component producer, which performs it once for all. A certification of this fact may be produced by some certification authority, in order for example to guarantee any recipient of a publicly available or migrating component that the component does not do anything different but what is described by its interfaces.

The verification of interface compatibility should instead be performed when the component is bound to another (e.g. at run-time when dealing with migrating code, that is when a migrating component reaches its final destination). We have shown that this check can be performed very efficiently by means of standard finite state space verification techniques. The higher complexity of checking conformance of components to their declared interface is left to an off-line verification activity, which may even need the use of infinite-state space verification techniques.

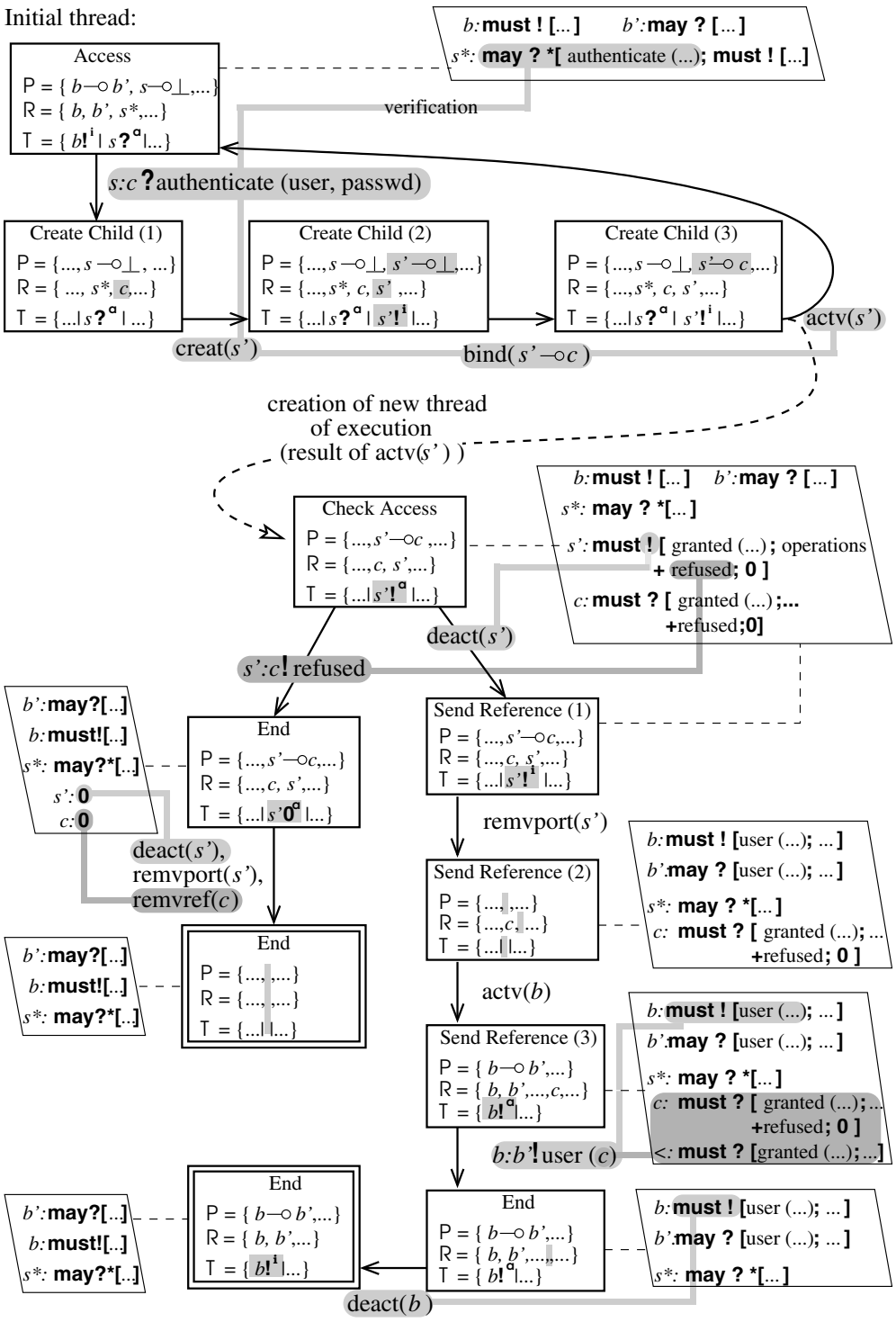


Figure 4. *Specifi cation and verifi cation of the* `Bank` *component*

We have applied our approach to a toy example; we need to verify the usability of the approach in practice, especially with respect to the expressiveness of the interface language we have proposed. Mixing modalities and actions (send/receive) is under investigation at NTNU, and may lead to some constraints like the ones proposed in [Flo03]. The conformance of the component model we have assumed with concrete notations (e.g. Java) should also be studied: varying the component model to suite a concrete notation may actually affect the classes of properties that can be guaranteed. We can also observe that the compatibility rules can be expressed in terms of temporal logic formulae: this would make it possible to prove in a logical framework a richer set of properties.

Acknowledgements C. Carrez and E. Najm have been partially supported by the RTNL ACCORD project and by the IST MIKADO project[6]. A. Fantechi has been partially funded by the 5% SP4 project of the Italian Ministry of University and Research. E. Najm has also been partially supported by a grant from ISTI of the Italian National Research Council.

Special thanks to Arnaud Bailly for his helpful advices.

8. References

[Car03] C. Carrez. *Contrats Comportementaux pour Composants*. PhD thesis, ENST, Paris, France, December 2003.

[CFN03a] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound composition of components. In *Formal Techniques for Networked and Distributed Systems (FORTE 2003, IFIP TC 6/WG 6.1)*, volume 2767 of *LNCS*. Springer-Verlag, Berlin, Germany, September 2003.

[CFN03b] C. Carrez, A. Fantechi, and E. Najm. Contrats comportementaux pour un assemblage sain de composants. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 2003)*, Paris, France, October 2003. French version of [CFN03a].

[dAH01] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-01*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*. ACM Press, 2001.

[Flo03] J. Floch. *Towards Plug-and-Play Services: Design and Validation using Roles*. PhD thesis, NTNU, Trondheim, Norway, February 2003.

[HR02] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *INFCTRL: Information and Computation (formerly Information and Control)*, 173, 2002.

[Kob02] N. Kobayashi. A type system for lock-free processes. *INFCTRL: Information and Computation (formerly Information and Control)*, 177, 2002.

[KPT99] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5), 1999.

[LSW95] K.G. Larsen, B. Steffen, and C. Weise. A constraint oriented proof methodology based on modal transition sytems. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS'95*, volume 1019 of *LNCS*, 1995.

[Nie95] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[NNS99] E. Najm, A. Nimour, and J.-B. Stefani. Infi nite types for distributed objects interfaces. In *Proc. of Formal Methods for Open Object-based Distributed Systems - FMOODS'99*, Firenze, Italy, February 1999. Kluwer.

6. Resp. `http://www.infres.enst.fr/projets/accord` and `http://mikado.di.fc.ul.pt/`