A Modal Logic for KLAIM

Rocco De Nicola and Michele Loreti

Dipartimento di Sistemi e Informatica, Università di Firenze
e-mail: {denicola,loreti}@dsi.unifi.it

Abstract. 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, and on allocation environments that associate logical localities to physical sites. This paper presents a temporal logic for specifying properties of Klaim programs. The logic is inspired by Hennessy-Milner Logic (HML) and the $\nu-$calculus, but has novel features that permit dealing with state properties to describe the effect of actions over the different sites. The logic is equipped with a consistent and complete proof system that enables one to prove properties of mobile systems.

Keywords: Mobile Code Languages, Temporal Logics of Programs, Coordination Models.

1 Introduction

The increasing use of wide area networks, especially the World Wide Web, is calling for new programming paradigms and for new programming languages that model interactions among clients and servers by means of *mobile agents*; these are programs that are transported and executed on different hosts. KLAIM (*a Kernel Language for Agents Interaction and Mobility*) [7] is one of such.

KLAIM consists of core Linda [3, 4] with multiple located tuple spaces and of a set of process operators, borrowed from Milner's CCS [6]. The underlying communication model is based on shared data space and is, thus, asynchronous.

In KLAIM, tuple spaces and processes are distributed over different localities, which are considered as first–class data. The classical Linda operations, indexed with the locations of the tuple space they operate on, allow programmers to distribute/retrieve data and processes over/from different nodes directly. Thus, programmers can directly manage the physical distribution of processes, the allocation policies, and the agents' mobility.

For KLAIM's programs, like for other class of programs, it is crucial to establish correctness, deadlock freeness, liveness and to control access rights. Since the language is based on process algebras, a natural candidate for such tasks is a temporal logic based on HML, the logic proposed by Hennessy and Milner to specify and verify properties of CCS agents [5].

However, one soon realizes that HML would be insufficient. For achieving our task we need both state formulae (to test for the presence of specific tuples at given localities) and richer actions (to specify the performed actions and their source and target).

In this paper we shall introduce a variant of HML with recursion, the syntax of our logic is then the following:

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

where the state properties are specified by the basic operator $t@\sigma$, and the classical indexed *diamond* operator ($\langle a\rangle$) is replaced by an action operator that contains sets of (abstract version of) the rich transition labels that are generated by the following grammar:

$$a ::= \mathcal{O}(s_1, t, s_2) \;\mid\; \mathcal{I}(s_1, t, s_2) \;\mid\; \mathcal{R}(s_1, t, s_2) \;\mid\; \mathcal{E}(s_1, P, s_2) \;\mid\; \mathcal{N}(s_1, -, s_2).$$

In the syntax above the label indicates source and destination of information movement (s_1 and s_2), the information transmitted (et and P) and the kind of movement ($\mathcal{O}$, $\mathcal{I}$,...).

Via abstract actions we can specify sets of labels that are characterized by common aspects, as source or destination of information movement, structure of the information transmitted and kind of movement.

We will show, via two simple examples, that the proposed logic is sufficiently expressive for describing interesting properties of mobile systems.

To support verification of such properties we will introduce also a proof system based on tableau. The proof system is inspired by [2], the additional difficulties and the novelties of our contribution are due to the fact that Cleaveland's system does not consider value passing and restricts attention to systems with a finite state space.

The rest of the paper is organized as follows. Section 2 contains the new labeled semantics for KLAIM. Section 3 contains syntax and semantics of the proposed logics together with its proofs system and a sketch of the soundness and completeness proof. Section 4 contains the KLAIM program for a distributed information system manager and the logical specification of some of its key properties. Section 5 shows a new equivalence that is in full agreement with the new one induced by the proposed logics.

Due to space limitation most of the proofs are omitted; they can be retrieved at [8]

2 KLAIM: syntax and semantics

KLAIM (Kernel Language for Agent Interaction and Mobility) is a language designed for programming applications over wide area networks. KLAIM is based on the notion of *locality* and relies on a Linda-like communication model.

Linda [1, 3, 4] is a coordination language with asynchronous communication and shared memory. Messages are structured data named *tuples*. The shared space is named *Tuple Space*. Tuples are accessed by pattern matching.

A KLAIM system is a set of *nodes* that we call *physical names* or *sites*. We use $\mathcal{S}$ to denote the set of *sites* and s, s_1, s_2, ... to denote its element.

Programs refer to sites using *localities*, or *logical name*. We use *Loc* to denote the set of localities and l, l_1, l_2, ... to denote its elements. We also assume existence of a locality $\texttt{self} \in Loc$. We will use ℓ to denote elements of $\mathcal{S} \cup Loc$. The operations over tuple spaces take as argument the name of a node where the target tuple space resides and a tuple.

Every node has a computational component, a set of processes running in parallel, a tuple space and an associated environment ρ that binds localities to sites. We also assume that in the node s the environment ρ is such that $\rho(\texttt{self}) = s$; i.e. the locality `self` refers to the node where a processes is running.

The set Net of KLAIM net is defined in Table 1. A node is defined by three parameters: the physical name s, the environment ρ and the process P. A net N can be obtained from the parallel composition of nodes.

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

Table 1. Nets syntax

For defining the syntax of processes, we introduce the following syntactic categories. We use Exp for the set expressions and Ψ for the set of parameterized *processes identifiers*, respectively ranged over by e and A. We use $VLoc$, Var and $VProc$ as the sets of locality, value and process variables, they are ranged over by u, x and X respectively. Moreover, $\widetilde{u}$ will indicate sequences of locality variables and $\{\widetilde{u}\}$ the set of locality variables in $\widetilde{u}$. A similar notation we will also be used for other kinds of sequences.

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

Table 2. Processes Syntax

Process syntax is defined in Table 2, where **nil** stands for the process that cannot perform any actions, $P_1|P_2$ stands for the parallel composition of P_1 and P_2, and $act.P$ stands for the process that executes the action act then behaves

like P. Also tuples are modeled as basic processes, then a tuple et is in the tuple space of a node s if and only if s contains a process $\mathbf{out}(et)$.

The possible actions are: $\mathbf{out}(t)@\ell$, $\mathbf{in}(t)@\ell$, $\mathbf{read}(t)@\ell$, $\mathbf{eval}(P)@\ell$ and $\mathbf{newloc}(u)$. The first action adds the result of evaluation of t, using the allocation environment ρ of the node where the action is performed, inside the tuple space of the site $\rho(l)$ (if it exists). If t is a tuple and ρ an environment we define the evaluation of t within the environment ρ, $\mathcal{T}[\![\, t \,]\!]_\rho$, as in Table 3. The operation $\mathbf{out}(t)@\ell$ is nonblocking.

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

Table 3. Tuple Evaluation Function

To retrieve information from a tuple space located at ℓ one can use the $\mathbf{in}(t)@\ell$ and $\mathbf{read}(t)@\ell$ primitives, differently from **out** these are blocking operations (i.e. the computation is blocked until the required action can be performed).

$match(v, v)$	$match(P, P)$	$match(s, s)$
$match(!\, x, v)$	$match(!\, X, P)$	$match(!\, u, s)$
$\dfrac{match(et_2, et_1)}{match(et_1, et_2)}$	$\dfrac{match(et_1, et_2) \quad match(et_3, et_4)}{match((et_1, et_3), (et_2, et_4))}$	

Table 4. The Matching Rules

The $\mathbf{in}(t)@\ell$ action looks for a tuple inside the tuple space at ℓ that satisfies the *matching predicate* defined in Table 4. If this tuple et exists then it is removed from the tuple space and the continuation process P is closed with respect to the substitution $[et/t]$ that replaces every variable in a formal field of t with the corresponding value in et.

The **read** operation behaves like **in** but it doesn't remove the tuple. Actions $\mathbf{in}(t)@\ell.P$ and $\mathbf{read}(t)@\ell.P$ act as binders for variables in the formal fields of t. A variable is *free* if and only if it isn't bound. We said that a process P is closed if and only if each variable in P is not *free*. From now on we will take in account only closed processes.

The primitive $\mathbf{eval}(P)@\ell$ spawns a process P at the site ℓ. The localities in P are evaluated with the allocation environment of the destination node.

The action $\mathbf{newloc}(u)$ creates a new node and binds the variable u to its new/fresh *name* s. The continuation process is closed with respect to the substitution $\{s/u\}$. Prefix $\mathbf{newloc}(u).P$ binds the locality variable u in P. Programmers, by means of **newloc** operations, can create private spaces.

Process identifiers are used in recursive process definitions. It is assumed that each process identifiers A has a *single* defining equation $A\langle \widetilde{X}, \widetilde{u}, \widetilde{x} \rangle$ and that all

free (values, processes or localities) variables in P are contained in $\{\widetilde{X}, \widetilde{u}, \widetilde{x}\}$. We also assume that all occurrences of process identifiers in P are guarded (i.e., each process identifier occurs within the scope of a blocking **in**/**read** prefix).

2.1 Operational Semantics

The evolution of a KLAIM net is described by singling out the tuples that are inserted, withdrawn or read from each node, or the processes that are spawned to other sites or the new/fresh sites that are created.

Example 1. Consider the net

$$N_1 = s_1 ::_{\rho_1} \mathbf{out}(t)@s_2.\mathbf{nil} || s_2 ::_{\rho_2} \mathbf{nil}$$

after placing the result of evaluating tuple t ($et = \mathcal{T}[\![t]\!]\rho$) on s_2, it evolves to the net

$$N_2 = s_1 ::_{\rho_1} \mathbf{nil} || s_2 ::_{\rho_2} \mathbf{out}(et)$$

$$\begin{array}{rll}
\mathbf{nil}\{\rho\} & = & \mathbf{nil} \\
X\{\rho\} & = & X \\
(\mathbf{out}(t)@\ell.P)\{\rho\} & = & \mathbf{out}(t\{\rho\})@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{eval}(Q)@\ell.P)\{\rho\} & = & \mathbf{eval}(Q)@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{in}(t)@\ell.P)\{\rho\} & = & \mathbf{in}(t\{\rho\})@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{read}(t)@\ell.P)\{\rho\} & = & \mathbf{read}(t\{\rho\})@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{newloc}(u).P)\{\rho\} & = & \mathbf{newloc}(u).P\{\rho\} \\
(P_1 \mid P_2)\{\rho\} & = & P_1\{\rho\} \mid P_2\{\rho\} \\
A\langle\widetilde{P}, \widetilde{\ell}, \widetilde{e}\rangle\{\rho\} & = & P[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{e}/\widetilde{x}]\{\rho\} \qquad \text{if } A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} P \\
\\
e\{\rho\} & = & e \\
(\ell)\{\rho\} & = & \rho(\ell) \\
!\,x\{\rho\} & = & !\,x \\
(!\,u)\{\rho\} & = & !\,u \\
(!\,X)\{\rho\} & = & !\,X \\
(f, t)\{\rho\} & = & f\{\rho\}, t\{\rho\}
\end{array}$$

Table 5. Closure Laws

We use labeled transitions to describe the evolution of nets. These labels indicate source and destination of information movement, the information transmitted and the kind of movement. We define the set of transition labels, *Lab*, as follows:

$$a ::= \mathcal{O}(s_1, et, s_2) \mid \mathcal{I}(s_1, et, s_2) \mid \mathcal{R}(s_1, et, s_2) \mid \mathcal{E}(s_1, P, s_2) \mid \mathcal{N}(s_1, -, s_2)$$

and we use a, possibly indexed, to range over *Lab*.

$$\frac{et = \mathcal{T}[\![\, t \,]\!]_\rho \qquad s' = \rho(\ell) \qquad a = \mathcal{O}(s, et, s')}{s ::_\rho \mathbf{out}(t)@\ell.P \parallel s' ::_\rho P' \stackrel{a}{\succ\!\!\rightarrow} s ::_\rho P \parallel s' ::_\rho (P' \mid \mathbf{out}(et))}$$

$$\frac{s' = \rho(\ell) \qquad a = \mathcal{E}(s, Q, s')}{s ::_\rho \mathbf{eval}(Q)@\ell.P \parallel s' ::_\rho P' \stackrel{a}{\succ\!\!\rightarrow} s ::_\rho P \parallel s' ::_\rho (P' \mid Q)}$$

$$\frac{match(\mathcal{T}[\![\, t \,]\!]_\rho, et) \qquad s' = \rho(\ell) \qquad a = \mathcal{I}(s, et, s')}{s ::_\rho \mathbf{in}(t)@\ell.P \parallel s' ::_\rho \mathbf{out}(et) \stackrel{a}{\succ\!\!\rightarrow} s ::_\rho P[et/\mathcal{T}[\![\, t \,]\!]_\rho] \parallel s' ::_\rho \mathbf{nil}}$$

$$\frac{match(\mathcal{T}[\![\, t \,]\!]_\rho) \qquad s' = \rho(\ell) \qquad a = \mathcal{R}(s, et, s')}{s ::_\rho \mathbf{read}(t)@\ell.P \parallel s' ::_\rho \mathbf{out}(et) \stackrel{a}{\succ\!\!\rightarrow} s ::_\rho P[et/\mathcal{T}[\![\, t \,]\!]] \parallel s' ::_\rho \mathbf{out}(et)}$$

$$\frac{s' \neq s \qquad a = \mathcal{N}(s, -, s')}{s ::_\rho \mathbf{newloc}(u).P \stackrel{a}{\succ\!\!\rightarrow} s ::_\rho P[s'/u] \parallel s' ::_\rho \mathbf{nil}}$$

$$\frac{s ::_\rho P[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{e}/\widetilde{x}] \stackrel{a}{\succ\!\!\rightarrow} N}{s ::_\rho A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e} \rangle \stackrel{a}{\succ\!\!\rightarrow} N} \; A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \stackrel{def}{=} P$$

$$\frac{N_1 \stackrel{a}{\succ\!\!\rightarrow} N_2 \qquad a \neq \mathcal{N}(s_1, -, s_2)}{N_1 \parallel N \stackrel{a}{\succ\!\!\rightarrow} N_2 \parallel N}$$

$$\frac{N_1 \stackrel{a}{\succ\!\!\rightarrow} N_2 \qquad a = \mathcal{N}(s_1, -, s_2) \qquad s_2 \notin N}{N_1 \parallel N \stackrel{a}{\succ\!\!\rightarrow} N_2 \parallel N} \qquad \frac{N_1 \equiv N_2 \qquad N_1 \stackrel{a}{\succ\!\!\rightarrow} N}{N_2 \stackrel{a}{\succ\!\!\rightarrow} N}$$

Table 6. The Operational Semantics

In Example 1 the label is $a = \mathcal{O}(s_1, t, s_2)$.

We use: $s \in N$ to denote that there exists a site named s in the net N; $s_\rho \in N$ if $s \in N$ and the allocation environment of s is ρ; $s_\rho :: P$ if $s_\rho \in N$ and P is running on s.

The operational semantics of KLAIM is given in Table 6. Where $\equiv$ is the structural congruence defined as the least congruence relation R such that:

$$(N_1 \parallel N_2)\ R\ (N_2 \parallel N_1),$$
$$((N_1 \parallel N_2) \parallel N_3)\ R\ (N_1 \parallel (N_2 \parallel N_3)),$$
$$(s ::_\rho^\delta (P_1 \mid P_2))\ R\ (s ::_\rho^\delta P_1 \parallel s ::_\rho^\delta P_2).$$

It easy to prove that this new labeled operational semantics coincides with the previous operational semantics based on rewriting systems [7].

We also write $N \succ\!\!\rightarrow^* N'$ if and only if:

1. $N' = N$;
2. $\exists a, N'' :\ N \stackrel{a}{\succ\!\!\rightarrow} N''$ and $N'' \succ\!\!\rightarrow^* N'$.

Example 2. In this example we analyze a Client-Server application. A client sends data to be evaluated by the server. The server evaluates them and sends back the result to the client. We have two sites, one for the client, and the other for the server.

At the server site, named s_S, there is a process that is waiting for a tuple containing two expressions and a site name. When such a tuple is present, the server returns the sum of the values to the site and restarts.

At the client site, named s_C, there is a process that sends, to the server sites, the tuple $(3, 5, \texttt{self})$ and waits for the result.

The KLAIM net for this system is:

$$\begin{array}{l} s_C ::_{\rho_C} \mathbf{out}(3, 5, \texttt{self})@server.\mathbf{in}(!result)@\texttt{self}.\mathbf{nil} \\ \quad \| \\ s_S ::_{\rho_S} ProcServer \end{array}$$

$ProcServer$ is defined as follow.

$$ProcServer \stackrel{def}{=} \mathbf{in}(!x_1, !x_2, !u)@\texttt{self}.\mathbf{out}(x_1 + x_2)@u.ProcServer$$

The evolution of the net start with the insertion of tuple $(3, 5, s_C)$ by the client in the tuple space of s_S (*label* $\mathcal{O}(s_C, (3, 5, s_C), s_S)$). Then process $ProcServer$ in s_S first removes tuple $(3, 5, s_C)$ (*label* $\mathcal{I}(s_S, (3, 5, s_C), s_S)$), then inserts tuple (8) in the tuple space of s_C (*label* $\mathcal{O}(s_S, (8), s_C)$). Finally tuple (8), is removed from s_C (*label* $\mathcal{I}(s_C, (8), s_C)$).

3 A Logic for KLAIM

We now introduce a logic that allows us to specify and prove properties of mobile system specified in KLAIM. In our view the important features of a KLAIM system are the tuples residing at specific nodes and the actions that a system performs during its evolution.

Our logic aims at capturing these two aspects. It permits to specify the presence of a tuple et inside the tuple space of a node s, by means of the atomic formula $et@s$, and the possible evolutions by means of the modal operators $\langle\cdot\rangle$, indexed by sets of actions.

3.1 Syntax

We use σ as a generic element in $\mathcal{S} \cup VLoc$. We also use $\mathbb{VAR}$ for $VLoc \cup Var \cup VProc$ and its elements are denoted with id, while $\mathbb{VAL}$ stands for $Val \cup Proc \cup \mathcal{S}$ and its elements are ranged by v.

To denote sets of actions that a KLAIM system can perform, we define the set of *abstract actions* $ALab$. An abstract action α is defined as follows:

$$\alpha ::= \mathcal{O}(\sigma_1, t, \sigma_2) \mid \mathcal{I}(\sigma_1, t, \sigma_2) \mid \mathcal{R}(\sigma_1, t, \sigma_2) \mid \mathcal{E}(\sigma_1, P, \sigma_2) \mid \mathcal{N}(\sigma_1, -, \sigma_2)$$

Obviously $Lab \subset ALab$.

Let $VLog$ be the set of logical variable ranged over by κ. We define $\mathcal{L}$ as the set of formulae ϕ obtainable by the following grammar:

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

where A is a subset of $ALab$. We shall also assume that no variable κ occurs negatively (i.e. under the scope of an odd number of $\neg$ operators) in ϕ.

We will use: $\langle\alpha\rangle\phi$ for $\langle\{\alpha\}\rangle\phi$, $\langle -\rangle\phi$ for $\langle Lab\rangle\phi$ and $\langle -A\rangle$ for $\langle Lab - \mathcal{A}[\![A]\!]\rangle\phi$.

We say that a variable id is *bound* in ϕ if every occurrence of id in ϕ appears in the scope of some $\langle A\rangle$ with $id \in \alpha$ for every $\alpha \in A$. A formula ϕ is closed if every variable in ϕ is bound.

Definition 1. *We define* $Subst \subseteq VLoc \to \mathcal{S} \uplus VProc \to VProc \uplus Var \to Val$, δ, *sometime with indexes, will be used to denote elements of* $Subst$.

If $\delta \in Subst$ and id is a variable then $\delta(id)$ is a value v of the same type of id. The closure of a formula ϕ with respect to a substitution δ ($\phi\{\delta\}$) is the formula ϕ' obtained from replacing *every variable* id in ϕ with $\delta(id)$. We also use $\delta_1 \cdot \delta_2$ for the substitution δ such that: $\delta(id) = \delta_2(id)$ if $\delta_2(id)$ is defined, $\delta(id) = \delta_1(id)$ otherwise.

3.2 Semantics

For specifying sets of actions that are characterized by common aspects, as source or destination of information movement, structure of the information transmitted and kind of movement, we use abstract actions.

Thus we first define the set of labels denoted by an abstract action α ($\mathcal{A}[\![\alpha]\!]$) as follows:

$$\mathcal{A}[\![\alpha]\!] = \{a | \exists\delta : a = \alpha\{\delta\}\}$$

i.e. $\mathcal{A}[\![\alpha]\!]$ is the set of action a such that there exists a substitution δ for which $a = \alpha\{\delta\}$; if $a \in \mathcal{A}[\![\alpha]\!]$ then we use δ^a_α for a δ' such that $\alpha\{\delta'\} = a$.

For example let $\alpha = \mathcal{I}(u, (''hello''), s)$ then

$$\mathcal{A}[\![\alpha]\!] = \{\mathcal{I}(s', (''hello''), s) | s' \in \mathcal{S}\}$$

and for $a = \mathcal{I}(s'', (''hello''), s) \in \mathcal{A}[\![\alpha]\!]$ we have that $\delta^a_\alpha = \{s''/u\}$.

Definition 2. *We define the* logical environment Env *as* $Env \subseteq [VLog \to Subst \to Net^*]$. *We also use* e, *sometime with indexes, to denote elements in* Env. *Moreover we use* $e \cdot [\kappa \mapsto g]$ *for the logical environment* e' *such that* $e'(\kappa') = e(\kappa')$ *if* $\kappa \neq \kappa'$, $e'(\kappa) = g$ *otherwise.*

We define $\mathcal{M}[\![\cdot]\!] : \mathcal{L} \to Env \to Subst \to Net^*$ to denote the set of nets that are models of a logical formula. Function $\mathcal{M}[\![\cdot]\!]$ is defined by structural induction as follows:

- $\mathcal{M}[\![\mathbf{tt}]\!]e\delta = Net$;
- $\mathcal{M}[\![\kappa]\!]e\delta = e(\kappa)\delta$
- $\mathcal{M}[\![t@\sigma]\!]e\delta = \{N | s = \sigma\{\delta\},\ et = t\{\delta\}, \exists\rho.\ s ::_\rho \mathbf{out}(et) \in N\}$;
- $\mathcal{M}[\![\langle\alpha\rangle\phi]\!]^H e\delta = \{N |\ \exists a \exists N' : N \overset{a}{\rightarrowtail} N' \wedge a \in \mathcal{A}[\![\alpha\{\delta\}]\!] \wedge N' \in \mathcal{M}[\![\phi]\!]^H e\delta \cdot \delta^a_{\alpha\{\delta\}}\}$;
- $\mathcal{M}[\![\langle A_1 \cup A_2\rangle\phi]\!]e\delta = \mathcal{M}[\![\langle A_1\rangle\phi]\!]e\delta \cup \mathcal{M}[\![\langle A_2\rangle\phi]\!]e\delta$

- $\mathcal{M}[\![\phi_1 \vee \phi_2]\!]e = \mathcal{M}[\![\phi_1]\!]e\delta \cup \mathcal{M}[\![\phi_2]\!]e\delta$;
- $\mathcal{M}[\![\neg\phi]\!]e\delta = Net - \mathcal{M}[\![\phi]\!]e\delta$;
- $\mathcal{M}[\![\nu\kappa.\phi]\!]e\delta = \nu f^{\phi}_{\kappa,e}\delta$ where:
  1. $f^{\phi}_{\kappa,e} : [Subst \to Net^*] \to [Subst \to Net^*]$ is defined as follows:
  $$f^{\phi}_{\kappa,e}(g) = \mathcal{M}[\![\phi]\!]e \cdot [\kappa \mapsto g]$$
  2. $\nu f^{\phi}_{\kappa,e} = \bigcup\{g | g \subseteq f^{\phi}_{\kappa,e}(g)\}$ where $g_1 \subseteq g_2$ if and only if for all δ $g_1(\delta) \subseteq g_2(\delta)$.

Other formulae like $[A]\phi$, $\mu\kappa.\phi$ or $\phi_1 \wedge \phi_2$ can be expressed with formulae in $\mathcal{L}$. Indeed $[A]\phi = \neg\langle A\rangle\neg\phi$, $\mu\kappa.\phi = \neg\nu\kappa.\neg\phi[\neg\kappa/\kappa]$ and $\phi_1 \wedge \phi_2 = \neg(\phi_1 \vee \phi_2)$.

Definition 3. *Let N be a net and ϕ be a closed formula, we say that N is a model of ϕ, written $N \models \phi$, if and only if $N \in \mathcal{M}[\![\phi]\!]e_0\delta_0$, where $e_0 = \lambda\kappa.\delta_0$ and $\delta_0 = \emptyset$.*

Example 3. If we consider the *Client/Server* application of Example 2, a property that we would like specify/verify is that if the tuple (x_1, x_2, u) is sent to the server then the tuple $(x_1 + x_2)$ is sent to the locality u from the server. This property can be specified with the formulae:

$$\begin{aligned}\phi = {} & \neg\nu\kappa.\neg(\langle\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle(\phi_1) \vee \neg(\ \neg\langle -\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\mathbf{tt} \\ & \qquad\qquad \vee\langle -\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\neg\kappa)) \\ \phi_1 = {} & \neg\nu\kappa'.\neg(\langle\mathcal{O}(u_2,(x_1+x_2),u_1)\rangle\mathbf{tt} \vee \neg(\ \langle -\mathcal{O}(u_2,(x_1+x_2),u_1)\rangle\mathbf{tt} \\ & \qquad\qquad \vee\neg\langle -\mathcal{O}(u_2,(x_1+x_2),u_1)\rangle\neg\kappa'))\end{aligned}$$

3.3 The proof system

We now introduce a tableau based proof system for $\mathcal{L}$ formulae. This proof system is based on [2] where a tableau-based system for μ-calculus has been introduced.

The proof rules operate on *sequents* of the form $H \vdash N : \phi$, where H is a set of *hypothesis* of the form $N' : \phi'$, N is a net, and ϕ is a closed formula. More correctly we should have written $H \vdash_{Net} N : \phi$, because we interpret N over Net, we omit the annotation for the sake of simplicity. We will refer to sequents by π and to proofs by Π.

If ϕ_1 and ϕ_2 are 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:

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. $\phi_2 = \langle A\rangle\phi_1$;
4. $\phi_2 = \nu\kappa.\phi_1$.

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

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

$$\frac{H \vdash N : \phi}{H \vdash N : \neg\neg\phi}\, R3 \qquad \frac{H \vdash N' : \phi\{\delta^a_\alpha\}}{H \vdash N : \langle A\rangle\phi}\, R4 - \left[N \succ\!\!\xrightarrow{a} N', \alpha \in A, a \in \mathcal{A}[\![\alpha]\!]\right]$$

$$\frac{H \vdash N_1 : \neg\phi\{\delta^{a_1}_\alpha\} \quad H \vdash N_2 : \neg\phi\{\delta^{a_2}_\alpha\} \ \ldots}{H \vdash N : \neg\langle A\rangle\phi}\, R5 - \left[\begin{array}{l} \forall\, i\, N \succ\!\!\xrightarrow{a_i} N_i,\ a_i \in \mathcal{A}[\![\alpha_i]\!], \\ \alpha_i \in A \end{array}\right]$$

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

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

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

Table 7. The proof system

Definition 4.

1. *A sequent $H \vdash N : \phi$ is* successful *if*
   - $\phi = \mathbf{tt}$.
   - $\phi = \nu\kappa.\phi'$ *and* $N : \nu\kappa.\phi' \in H$;
   - $\phi = \neg\langle A\rangle\phi'$, *and* $\nexists a \in \mathcal{A}[\![A]\!]$ *such that* $N \succ\!\!\xrightarrow{a} N'$;
   - $\phi = et@s$ *and* $s ::_\rho \mathbf{out}(et) \in N$;
   - $\phi = \neg et@s$ *and* $s ::_\rho \mathbf{out}(et) \notin N$;
2. *Π is a successful proof for π if the following conditions hold:*
   - *Π is built using the rules on Table 7;*
   - *π is the root of Π;*
   - *every leaf on Π is a successful sequent.*
3. *π is provable if and only if there exists a successful proof Π for π.*

We define the models of a formula ϕ with the hypothesis H, $\mathcal{M}[\![\phi]\!]^H$, as follows:

- $\mathcal{M}[\![\mathbf{tt}]\!]^H e\delta = Net$;
- $\mathcal{M}[\![\kappa]\!]^H e\delta = e(\kappa)\delta$
- $\mathcal{M}[\![t@\sigma]\!]^H e\delta = \{N | s = \sigma\{\delta\},\ et = t\{\delta\},\ s ::_\rho \mathbf{out}(et) \in N\}$;
- $\mathcal{M}[\![\langle\alpha\rangle\phi]\!]^H e\delta = \{N | \exists a \exists N' : N \succ\!\!\xrightarrow{a} N' \wedge a \in \mathcal{A}[\![\alpha\{\delta\}]\!] \wedge N' \in \mathcal{M}[\![\phi]\!]^H e\delta \cdot \delta^a_{\alpha\{\delta\}}\}$;
- $\mathcal{M}[\![\langle\alpha\rangle\phi]\!]^H e\delta = \{N | \exists a \exists N' : N \succ\!\!\xrightarrow{a} N' \wedge a \in \mathcal{A}[\![\alpha\{\delta\}]\!] \wedge N' \in \mathcal{M}[\![\phi]\!]^H e\delta \cdot \delta^a_{\alpha\{\delta\}}\}$;
- $\mathcal{M}[\![\phi_1 \vee \phi_2]\!]^H e\delta = \mathcal{M}[\![\phi_1]\!]^H e\delta \cup \mathcal{M}[\![\phi_2]\!]^H e\delta$;
- $\mathcal{M}[\![\neg\phi]\!]^H e\delta = Net - \mathcal{M}[\![\phi]\!]^H e\delta$;
- $\mathcal{M}[\![\nu\kappa.\phi]\!] e\delta = \nu f^{\phi,h}_{\kappa,e}\delta \cup h\delta$ where:
  1. $f^{\phi,h}_{\kappa,e} : [Subst \to Net^*] \to [Subst \to Net^*]$ is defined as follows:

$$f^{\phi,h}_{\kappa,e}(g) = f^{\phi}_{\kappa,e}(g \cup h)$$

2. $h : Subst \to Nets^*$ is defined as follows:

$$h\delta = \{N | N : \nu\kappa.\phi\{\delta\} \in H\}$$

3. $\nu f^{\phi,h}_{\kappa,e} = \bigcup\{g | g \subseteq f^{\phi,h}_{\kappa,e}(g)\}$.

If $H = \emptyset$ then $\mathcal{M}[\![\phi]\!]^H e\delta = \mathcal{M}[\![\phi]\!]e\delta$.

Definition 5. *Let N be a net, and let ϕ be a closed formula, we say that N is a model of ϕ under the hypothesis H, written $N \models_H \phi$, if and only if $N \in \mathcal{M}[\![\phi]\!]^H e_0\delta_0$, with $e_0 = \lambda\kappa.\delta_0$ and $\delta_0 = \emptyset$.*

Theorem 1. *If there exists a proof Π for $H \vdash N : \phi$ then $N \models_H \phi$.*

Theorem 2. *Let N be such that the set $\{N' | N \succ\!\!\rightarrow^* N'\}$ is finite then, for all closed formula ϕ, $N \models_H \phi$ implies $H \vdash N : \phi$ provable.*

Theorem 3. *Let ϕ be a closed formula such that:*

- *if $\nu\kappa.\phi'$ is a subformula of ϕ then it is negative in ϕ;*
- *if $\langle A\rangle\phi'$ of ϕ is such that, if there exists $\alpha = \mathcal{N}(\sigma, -, u) \in A$, then $\langle A\rangle\phi'$ is no negative in ϕ;*

then for all net N and for all set of hypothesis H if $N \models_H \phi$ then $H \vdash N : \phi$ is provable.

Example 4. We want now to show how, using the proof system, we can prove that system CS of Example 2 satisfies formula ϕ of Example 3.

Thus we want prove that sequent $\emptyset \vdash CS : \phi$ is provable, i.e. there exists a proof for it. Now the only rule that we can apply to the sequent is $R7$. Thus we start our proof as follows:

$$\frac{CS : \nu\kappa.\phi' \vdash CS : \begin{array}{r}\neg\neg(\langle\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle(\phi_1) \vee \neg(\ \neg\langle -\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\mathbf{tt} \\ \vee\langle -\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\neg\nu\kappa.\phi'))\end{array}}{\emptyset \vdash CS : \phi}$$

where

$$\phi' = \neg(\langle\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle(\phi_1) \vee \neg(\ \neg\langle -\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\mathbf{tt} \\ \vee\langle -\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\neg\kappa))$$

We can now proceed by applying rules $R3$ and $R1$ obtaining:

$$CS : \nu\kappa.\phi' \vdash CS : \langle\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\phi_1$$

Net CS can only evolve, by action $\mathcal{O}(s_C,(3,5,s_C),s_S)$, to

$$\begin{array}{rl} SC' = s_C & ::_{\rho_C} \mathbf{in}(!result)@\mathtt{self}.\mathbf{nil} \\ \| & \\ s_S & ::_{\rho_S} \mathbf{in}(!x_1,!x_2,!u)@\mathtt{self}.\mathbf{out}(x_1+x_2)@u.ProcServer \\ & \quad |\mathbf{out}(3,5,s_C) \end{array}$$

then applying rule $R4$ we have that:

$$\frac{CS : \nu\kappa.\phi' \vdash CS' : \begin{array}{r}\neg\nu\kappa'.\neg(\langle\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt} \vee \neg(\ \neg\langle -\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt} \\ \vee\langle -\mathcal{O}(s_S,(8),s_C)\rangle\neg\kappa'))\end{array}}{CS : \nu\kappa.\phi' \vdash CS : \langle\mathcal{O}(u_1,(x_1,x_2,u),u_2)\rangle\phi_1}$$

Let ϕ_1' be such that

$$\phi_1' = \begin{array}{r}\neg(\langle\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt} \vee \neg(\ \neg\langle -\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt} \\ \vee\langle -\mathcal{O}(s_S,(8),s_C)\rangle\neg\kappa'))\end{array}$$

then by rule $R7$ we have that

$$\frac{CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1' \vdash CS' : \neg\phi_1'[\nu\kappa'.\phi'/\kappa']}{CS : \nu\kappa.\phi' \vdash CS' : \neg\nu\kappa'.\phi_1'}$$

As in a previous case, applying rules $R3$ and $R1$, we obtain the sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1' \vdash CS' : \neg(\neg\langle -\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt} \vee \langle -\mathcal{O}(s_S,(8),s_C)\rangle\neg\nu\kappa'.\phi_1')$$

applying $R2$ we have to prove sequents:

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1' \vdash CS' : \neg\neg\langle -\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt} \quad (1)$$

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1' \vdash CS' : \neg\langle -\mathcal{O}(s_S,(8),s_C)\rangle\neg\nu\kappa'.\phi_1' \quad (2)$$

Net CS' can only evolve, by an action $\mathcal{I}(s_S,(3,5,s_C),s_S)$, to the net:

$$CS'' = s_C ::_{\rho_C} \mathbf{in}(!result)@\mathtt{self}.\mathbf{nil} \parallel s_S ::_{\rho_S} \mathbf{out}(3+5)@s_C.ProcServer$$

Then by rule $R3$ and $R4$, for (1), we obtain the successfully sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1' \vdash CS' : \mathbf{tt}$$

while for (2) we obtain, by rule $R5$, sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1' \vdash CS'' : \neg\nu\kappa'.\phi_1'$$

Applying rules $R7$, $R3$ and $R1$ again, we obtain the sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1', CS'' : \nu\kappa'.\phi_1' \vdash CS'' : \langle\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt}$$

Net CS'' evolves, by $\mathcal{O}(s_S,(8),s_C)$, to the net

$$CS''' = s_C ::_{\rho_C} \mathbf{in}(!result)@\mathtt{self}.\mathbf{nil}|\mathbf{out}(8) \parallel s_S ::_{\rho_S} ProcServer$$

thus, by rule $R4$, we have

$$\frac{CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1, CS'' : \nu\kappa'.\phi_1' \vdash CS''' : \mathbf{tt}}{CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1', CS'' : \nu\kappa'.\phi_1' \vdash CS'' : \langle\mathcal{O}(s_S,(8),s_C)\rangle\mathbf{tt}}$$

hence we have obtained a proof of $\emptyset \vdash CS : \neg\nu\kappa.\phi'$.

4 An extended example

In this section we consider a larger example of a Distribute Information System management.

We assume that a Database system is distributed over three different sites, named Inf_i ($i \in \{1,2,3\}$). A node, named $Manager$, manages the database system sending processes for updating the information on the nodes. The updating process chooses a path to reach every node. Only one updating-process at a time can be executed in a site. For this reason inside the tuple space of Inf_i there is the tuple $''F''$. An updating process can be evaluated in an Inf_i node only when tuple $''F''$ is in its tuple space.

The net of the distributed database is defined as follows:

$$Inf_1 :: \mathbf{out}(''F'') \parallel Inf_2 :: \mathbf{out}(''F'') \parallel Inf_3 :: \mathbf{out}(''F'')$$

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 tuple is found, the process $CallUpdate$, which starts the updating procedure, is called. Guarding $CallUpdate$ in $StartAgent$ with an $\mathbf{in}(''G'')$ we ensure that the system is deadlock free.

$$StartAgent = \mathbf{in}(''G'')@\mathtt{self}.\ (CallUpdate(Inf_1, Inf_2, Inf_3) \\ |StartAgent)$$

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

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

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

$$Success = \mathbf{out}(''G'')@\mathtt{self}.\mathbf{nil}$$

The manager node is define as follows:

$$Manager :: StartAgent|\mathbf{out}(Inf_1)|\mathbf{out}(Inf_2)|\mathbf{out}(Inf_3) \\ |\mathbf{out}(''G'')|\mathbf{out}(''G'')|\mathbf{out}(''G'')$$

For this system, we would like to specify that if a process $Update(s, P)$ (respectively $FUpdate(s)$) is evaluated in a site Inf_i, for some site s and some process P, then no processes are evaluated on Inf_i until process P (respectively $Success$) is evaluated from Inf_i to the site s. This property is specified with the following formulae:

$$\phi_1 = \neg\langle\mathcal{E}(u_1, Update(u_2, X), u_3)\rangle\neg(\nu\kappa_1.\ \langle\mathcal{E}(u_3, X, u_2)\rangle\mathbf{tt}\vee \neg(\langle\mathcal{E}(u_4, X', u_3)\rangle\mathbf{tt}\vee \langle-\mathcal{E}(u_3, X, u_2)\rangle\neg\kappa_1)$$

$$\phi_2 = \neg\langle\mathcal{E}(u_1, FUpdate(u_2), u_3)\rangle\neg(\nu\kappa_2.\ \langle\mathcal{E}(u_3, Success, u_2)\rangle\mathbf{tt}\vee \neg(\langle\mathcal{E}(u_4, X', u_3)\rangle\mathbf{tt}\vee \langle-\mathcal{E}(u_3, Success, u_2)\rangle\neg\kappa_2)$$

We wish that ϕ_1 and ϕ_2 was verified in every reachable state of our system. This is specified with the formula:

$$\phi = \nu\kappa.\neg((\neg\phi_1 \vee \neg\phi_2) \vee \neg\langle-\rangle\neg\kappa)$$

Due to space limitation we omit the proof for ϕ.

5 Behaviours of Nets

In this section we introduce a new equivalence relation between KLAIM nets and we will show as it is in full agreement with the one induced by the proposed logics.

Nets will be compared according to their action tree or *behaviour*. The behaviours of nets are generated respect the following syntax:

$$\Gamma ::= \bot \ \mid\ \omega \ \mid\ a \to \Gamma \ \mid\ \Gamma \wedge \Gamma \ \mid\ et@s$$

The set of all possible behaviour will be denoted by Γ.

We will write $N : \Gamma$ to indicate that the net N has the behavior Γ. A particular behaviour $\bot$ is introduced to represent *fully unspecified behaviour*; every net N has $\bot$ ($N : \bot$) as a possible behaviour. A net N has a behaviour $et@s$ if the tuple et is in the tuple spaces of the site s of N

The behaviour $a \to \Gamma$ represent the set of nets that are able to perform an action a and then behaves like Γ, so a net N has a behaviour $\Gamma = a \to \Gamma'$ if N' exists such that $N \succ\!\!\xrightarrow{a} N'$ and $N' : \Gamma$.

A net N has a behaviour $\Gamma_1 \wedge \Gamma_2$ if it has both Γ_1 and Γ_2 ($N : \Gamma_1 \wedge \Gamma_2$ if $N : \Gamma_1$ and $N_2 : \Gamma_2$).

The behaviour ω represent the capability of performing any actions; no net has behaviour ω.

Definition 6. *We say that $N : \Gamma$ if and only if we are able to prove that with the following rules:*

$$\frac{}{N : \bot} \qquad \frac{N \succ\!\!\xrightarrow{a} N' \qquad N' : \Gamma}{N : a \to \Gamma} \qquad \frac{N : \Gamma_1 \qquad N : \Gamma_2}{N : \Gamma_1 \wedge \Gamma_2}$$

To reason on behaviours we introduce an ordering between them.

Definition 7. $\leq$ *is the smallest relation defined as follows:*

- $\Gamma \leq \omega$
- $\bot \leq \Gamma$
- *if* $\Gamma_1 \leq \Gamma_2$ *then* $a \to \Gamma_1 \leq a \to \Gamma_2$
- *if* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_2 \leq \Gamma_3$ *then* $\Gamma_1 \leq \Gamma_3$
- $\Gamma_1 \wedge \Gamma_2 \leq \Gamma_2 \wedge \Gamma_1$
- $\Gamma_1 \leq \Gamma \wedge \Gamma_1$
- *if* $\Gamma_1 \leq \Gamma_2$ *then* $\Gamma_1 \wedge \Gamma \leq \Gamma_2 \wedge \Gamma$

If we interpret behaviours as requirements on computing agents then the ordering $\Gamma' \leq \Gamma$ indicates that a net with a behaviour Γ satisfies more requirements then a net with a behaviour Γ'.

In this point of view ω is the highest while $\bot$ is the lowest. If $N_1 : \Gamma_1 \wedge \Gamma_2$ then N_1 has both the behaviours Γ_1 and Γ_2 and if $N_2 : \Gamma_2$ then N_1 satisfies more requirements. So $\Gamma_2 \leq \Gamma_1 \wedge \Gamma_2$ and the operator $\wedge$ is commutative and associative.

Definition 8.

1. *We write* $N_1 \sqsubseteq N_2$ *if and only if for all* Γ_1 *if* $N_1 : \Gamma_1$ *then there exists* Γ_2*, with* $N_2 : \Gamma_2$*, such that* $\Gamma_1 \leq \Gamma_2$*;*
2. *We write* $N_1 \simeq N_2$ *if and only if* $N_1 \sqsubseteq N_2$ *and* $N_2 \sqsubseteq N_1$*.*

Theorem 4. *For all net* N_1*,* N_2 *we have that* $N_1 \simeq N_2$ *if and only if for all formula* $\phi \in \mathcal{L}$ $N_1 \models \phi$ *if and only if* $N_2 \models \phi$

References

1. Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(10):444–458, October 1989. Technical Correspondence.
2. Rance Cleaveland. Tableau-based model checking in the propositional μ-calculus. *Acta Informatica*, 27(8):725–747, September 1990.
3. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
4. D. Gelernter. Multiple tuple spaces in linda. In J.Hartmanis G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *LNCS*, pages 20–27, 1989.
5. Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.
6. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.
7. Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special issue: Mobility and Network Aware Computing.
8. Rocco De Nicola and Michele Loreti. A logic for klaim (full paper). Avaiable at `ftp://rap.dsi.unifi.it/papers/fullMLK.ps`.