Open nets, contexts and their properties

Rocco De Nicola Michele Loreti

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

Abstract. KLAIM (A Kernel Language for Agents Interaction an Mobility) is a simple formalism that can be used to model and program mobile systems; it allows programmers to concentrate on the distributed structure of programs while ignoring their precise physical allocations. For establishing properties of KLAIM Nets, a logic based on HML was defined. This logic permits specifying properties related to resource allocation, security and behaviours. A weakness of the approach is that, in order to use KLAIM and its associated logic to establish nets properties one needs to have a full implementation of the system under consideration. This is a very strong assumption when dealing with wide area networks, because very often, only a fragment of the system is known and a limited knowledge of the overall context is available. In the paper, after recalling KLAIM and its logic, we shall present a framework for specifying contexts for KLAIM. Then, by relying on a notion of agreement, we shall set up a framework ensuring that, whenever an abstract specification is partially implemented as a concrete net (guaranteeing agreement with specifications) all formulae satisfied by the more abstract description are satisfied also by the refined one.

1 Introduction

Many calculi [2, 14, 7, 11, 16] have been proposed for modelling systems distributed over Wide Area Network (WAN). Indeed, classical frameworks for concurrent and locally distributed systems, like variants of process algebras, do not guarantee *network awareness*. The new calculi are equipped with primitives for dealing with the distribution of resources and computational components, for handling code and agent mobility and for coordinating processes.

For WAN, like for concurrent systems, it is crucial to have tools for establishing deadlock freedom, 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 resources allocation, access to resources, agent movement, and information disclosure. For this purpose, some papers [3, 4, 10] advocate the use of temporal logics for specifying and verifying dynamic properties of mobile agents running over a wide area network. However, to use these formal tools to establish system properties detailed descriptions of the whole systems under consideration are required.

Obviously, this is a very strong assumption for wide area networks, because, very often only some components of the system are known; and one has only a limited knowledge of the overall context in which the component is operating.

Cardelli and Gordon [3] partially solve this problem by introducing the modal operator $\triangleright$ whose informal semantics is: let ϕ and ψ be two logical formulae, a system A satisfies $\phi \triangleright \psi$ if and only if **for each** system B satisfying ϕ the parallel composition of A with B satisfies ψ. The new operator permits establishing a connection between the properties satisfied by a system and those of its environment, but, due to the universal quantification that appears in the interpretation, the full logic is undecidable.

In this paper, after introducing a variant of KLAIM, we shall present a framework for the abstract specification of KLAIM contexts. Then, by relying on a notion of *agreement* of a net N with a context specification $C[\,]$, we shall set up a framework that guarantees that if a context specification is partially instantiated (implemented) as a concrete net, then all formulae satisfied by the more abstract description are satisfied also by the refined one.

More specifically we shall prove that if $C_1[N_1]$ satisfies a formula ϕ and $C_2[N_1 \parallel N_2]$ is a refinement of $C_1[N_2]$ then also $C_2[N_1 \parallel N_2]$ satisfies ϕ.

The provided framework permits modelling *partially specified* systems; moreover, thanks to the notion of *agreement* between *contexts* and *real implementations*, it permits using a stepwise refinement procedure for program development: the properties verified at some level of abstraction will be preserved after refinement.

The rest of the paper is organized as follows. Section 2 contains a brief introduction to μKLAIM and its labelled operational semantics; the modal logic is briefly presented in Section 3. In Section 4 a framework for specifying contexts over μKLAIM nets is defined while in Section 5 it is shown how contexts can be refined and how this procedure can guarantee preservation of formulae satisfaction. Section 6 concludes the paper. A simple running example will be used through the paper for illustrating our approach.

Due to space limitation most of the proofs are omitted; interested reader can retrieved them from [9].

2 μKLAIM

KLAIM [8] (A Kernel Language for Agents Interaction an Mobility) is a simple formalism that can be used to model and program mobile systems; it allows programmers to concentrate on the distributed structure of programs while ignoring their precise physical allocations. Primitives are designed to provide the programmer with the distributed infrastructure for handling physical distribution, scoping and mobility of processes. KLAIM is based on process algebras but makes use of Linda-like asynchronous communication and models distribution via multiple shared tuple spaces [5, 12, 13]. Tuple spaces and processes are distributed over different localities and the classical Linda operations are indexed with the location of the tuple space they operate on.

For establishing properties of KLAIM Nets, we have designed a logic that is based on HML but has richer action predicates that permit reasoning on the information transmitted over the net and has state formulae for testing the presence of specific tuples at given localities. We shall use our KLAIM logic as a basis to work out a logic of context and refinements that, by relying on a notion of behavioural equivalence of nets, will enable us to perform incremental construction and verification of KLAIM nets by exploiting a sort of rely-guarantee approach.

For the sake of simplicity, we shall use a simplified version of KLAIM: μKLAIM introduced in [1]. The main differences between KLAIM and μKLAIM being that the former allows high-level communication (processes can be used as tuple fields) while the latter only permits evaluating process remotely. Moreover, the simpler language does not make any distinction between physical and logical localities and does not rely on allocation environments.

A μKLAIM system, called a *net*, is a set of *nodes*, univocally associated to *localities* that can be seen as addresses used to refer to nodes. We shall use $\mathcal{L}$ to denote the set of localities $l, l_1, \ldots$ and $\mathcal{U}$ to denote the set of *locality variables* u,u_1,.... The symbol ℓ, possibly with subscripts, will be used to denote either localities or locality variables. Every node has a computational component (a set of processes running in parallel) and a data component (a tuple space). Processes interact via tuple spaces either locally or remotely.

The syntax of μKLAIM nets is defined in Table 1. A node is defined by two parameters. The first one, is the locality l that identifies the node. The second one is either the process P, running at l, or a tuple, which is stored in the tuple space ($\langle et \rangle$) located at l.

μKLAIM nets are obtained by parallel composition of nodes. We shall use $l \in N$ to denote that N contains a node whose locality is l. We use $loc(N)$ to denote the set of localities that belong to N.

Process syntax is defined in Table 1, 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**, that inserts the evalutaion of a tuple t ($\mathcal{T}[\![t]\!]$) in a located tuple space, **in** and **read**, which retrieve a located tuple matching a *template* T, and **eval** that spawns a process to be evaluated remotely[1].

[1] In the *standard* versions of μKLAIM, processes are also equipped with a primitive, named **newloc**, for creating new nodes with a new *fresh* locality. For the sake of simplicity, we do not cosider that here

$N ::=$		NETS	$P ::=$		PROCESSES
	$l :: P$	*single node*		**nil**	*Nil process*
	$l :: \langle et \rangle$	*located tuple*		$a.P$	*Action Prefixing*
	$N_1 \parallel N_2$	*net composition*		$P \mid P$	*Parallel Composition*
				$P + P$	*Nondeterministic Choice*
$T ::=$	$F \mid F,T$	TEMPLATES		X	*Recurcion Variable*
$F ::=$	$f \mid !x \mid !u$	TEMPLATE FIELDS		**rec**$X.P$	*Recursion*
$t ::=$	$f \mid f,t$	TUPLES			
$f ::=$	$e \mid l \mid u$	TUPLE FIELDS	$a ::=$		ACTIONS
$et ::=$	$ef \mid ef,et$	EVALUATED TUPLES		**out**$(t)@l$	*output*
$ef ::=$	$V \mid l$	EVALUATED FIELDS		**in**$(T)@l$	*input*
$e ::=$	$V \mid x \mid \ldots$	EXPRESSIONS		**read**$(T)@l$	*read*
				eval$(P)@l$	*migration*
				newloc(u)	*creation*

Table 1. μKLAIM Syntax

A tuple is a sequence of *actual* fields. Each actual field can be either a locality (l), a variable (x or u) or an expression (e). The syntax for expressions is not specified; we only assume that e could be a value v, from the set of *basic values Val*, or a *variable x*, from the set of *variables Var*. A tuple is evaluated, and called *et*, if all of its fields are either localities or basic values. Tuples are retrieved from tuple spaces via *pattern matching* using *templates* (T). Templates are sequences of *actual* and *formal* fields. these are *variables* that will get a value when a tuple is retrieved. Formal field are signalled by a '!' before the variable name.

$$(\mathrm{M}_1)\ \ match(V,V) = [] \qquad (\mathrm{M}_2)\ \ match(!x,V) = [^V/_x]$$
$$(\mathrm{M}_3)\ \ match(l,l) = [] \qquad (\mathrm{M}_4)\ \ match(!u,l) = [^l/_u]$$
$$(\mathrm{M}_5)\ \ \frac{match(F,f) = \sigma_1 \quad match(T,t) = \sigma_2}{match(\ (F,T)\ ,\ (f,t)\ ) = \sigma_1 \cdot \sigma_2}$$

Table 2. Matching Rules

The matching predicate is defined in Table 2: actual fields match if they are identical while formal fields match any field of the same type. A template T and a tuple t do match if they have the same number of fields and the corresponding fields do match. A successful matching returns a substitution function replacing the variables contained in the formal fields of the template with the values contained in the corresponding actual fields of the accessed tuple. We let *Subst* be the set of substitutions (with finite domain) σ,σ_1,.... We write $\cdot$ to denote substitutions composition and [] to denote the 'empty' substitution.

The **out**$(t)@\ell$ action adds the result of evaluating t to the tuple space at locality ℓ. Where, the evaluation of t ($\mathcal{T}[\![t]\!]$) is the tuple obtained from t replacing each expression with the corresponding value. The actions **in**$(T)@\ell$ and **read**$(T)\ell$ are used to retrieve information from tuple spaces. Differently from **out** these are blocking operations; the computation is blocked until the required action can be performed, i.e. a tuple matching template T can be found.

The **in**$(T)@\ell$ action looks for a tuple matching T in the tuple space located ad ℓ. When a matching tuple *et* is found it is removed from the tuple space and the continuation process is closed with substitution $\sigma = match(T,et)$ that replaces the formals in T with the corresponding values in *et*. For instance, if $T = (!u,4)$ and $et = (l,4)$ then $match(T,et) = [l/u]$. The **read** operation behaves like **in** but does not remove tuples. The actions **in**$(T)@\ell.P$ and **read**$(T)@\ell.P$ act as a binder for variables in the formal fields of T. The primitive **eval**$(P)@\ell$ spawns a process P at the locality ℓ.

A variable will be called *free* if and only if it is not bound. A process P is closed if and only all its variables are *bound*. In the rest of the paper, we shall only consider closed processes. We will write $P =_\alpha Q$ if processes P and Q are equal up to renaming of bound variables.

The operator **rec** $X.P$ is used to define recursive processes where we assume that each occurrence of X in P are guarded, i.e. under the scope of an action prefixing operator.

$$(\textsc{Out})\quad \frac{\mathcal{T}[\![t]\!] = et}{N \parallel l :: \mathbf{out}(t)@l'.P \parallel l' :: P' \succ\!\xrightarrow{l:\mathbf{o}(et)l'} N \parallel l :: P \parallel l' :: P' \parallel l' :: \langle et\rangle}$$

$$(\textsc{Eval})\quad N \parallel l :: \mathbf{eval}(Q)@l'.P \parallel l :: P' \succ\!\xrightarrow{l:\mathbf{e}(Q)@l'} N \parallel l :: P \parallel l' :: P'|Q$$

$$(\textsc{In})\quad \frac{match(\mathcal{T}[\![T]\!], et) = \sigma}{N \parallel l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle et\rangle \succ\!\xrightarrow{l:\mathbf{i}(et)@l'} N \parallel l :: P\sigma \parallel l' :: \mathbf{nil}}$$

$$(\textsc{Read})\quad \frac{match(\mathcal{T}[\![T]\!], et) = \sigma}{N \parallel l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle et\rangle \succ\!\xrightarrow{l:\mathbf{r}(et)@l'} N \parallel l :: P\sigma \parallel l' :: \langle et\rangle}$$

$$(\textsc{Struct})\quad \frac{N \equiv N_1 \qquad N_1 \succ\!\xrightarrow{\lambda} N_2 \qquad N_2 \equiv N'}{N \succ\!\xrightarrow{\lambda} N'}$$

where

$$N_1 \parallel N_2 \equiv N_2 \parallel N_1 \qquad (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$$
$$l :: P \equiv l :: (P|\mathbf{nil}) \qquad l :: \mathbf{rec}X.P \equiv l :: P[\mathbf{rec}X.P/X]$$
$$l :: (P_1|P_2) \equiv l :: P_1 \parallel l :: P_2$$

Table 3. μKLAIM Labelled Operational Semantics

To make the involved localities and the information transmitted over the net evident, we have to rely on a labelled operational semantics. Transition labels carry information about the action performed, the localities involved in the action, and the transmitted information. We consider Λ as the set of λ whose structure is $l_1 : a@l_2$. Locality l_1 denotes the node where the action is executed, l_2 is the node where the action takes effect, term a, which describes the kind of action performed and the information transmitted, has the following syntax:

$$a ::= \mathbf{o}(et) \mid \mathbf{i}(et) \mid \mathbf{r}(et) \mid \mathbf{e}(P)$$

In the rest of the paper we will refer to l_1 and l_2, respectively, as the source and the target of a transition label λ. For instance, if a process running at l_1 inserts $\mathcal{T}[\![t]\!]$ in the tuple space located at l_2, by executing $\mathbf{out}(t)@l_2$, then the corresponding transition of the net is labelled by $l_1 : \mathbf{o}(\mathcal{T}[\![t]\!])@l_2$.

In this paper, we shall define two kinds of transition relations: a classical one and a new one that make *invisible* all the actions that do not involve localities in a given set L.

Let $\succ\!\xrightarrow{\cdot}$ be the least relation induced by the rules in Table 3. Let L be a set of localities, the *reduction relation* $\succ\!\xrightarrow{\cdot}_L$ is defined as follows: for each N_1 and N_2, $N_1 \succ\!\xrightarrow{l_1:a@l_2}_L N_2$ if and only if:

- $N_1 \succ\!\xrightarrow{l_1:a@l_2}_L N_2$ and either l_1 or l_2 belongs to L;
- $N_1 \succ\!\xrightarrow{l_1':a'@l_2'} N'$, $\{l_1', l_2'\} \notin L$ and $N' \succ\!\xrightarrow{l_1:a@l_2}_L N_2$

Definition 1. *Let L be a set of localities, we say that a net N* leaves *L* forever *if there exists an infinite sets of nets $\{N_i\}$ such that $N_0 = N$ and for each i $\exists l_1, l_2$: $N_i \succ\!\xrightarrow{l_1:a@l_2} N_{i+1}$ and $l_1, l_2 \notin L$.*

This latter notion permits characterizing those nets with actions that either lead to stopping or to passing infinitevely often *through* the subnet singled out by L.

Example 1 (A PrintServer). Here we show how μKLAIM can be used for modelling a simple print server. In Table 4 you can find three μKLAIM nodes: *PrintServer*, *Printer* and *PClient*. Located at *PrintServer* there is a process that waits for a print request. Each request contains the locality that sent the request. When a request is found in the tuple space locate at *PrintServer*, the process sends to the printer the document and waits for the result that, when available, will be comunitated to the remote client. Each client, first retrieves

$$
\begin{array}{l}
\mathit{Printer} :: \mathbf{rec}X.\mathbf{in}(!\,\mathit{from})@\mathit{Printer}. \\
\qquad\qquad (X \,|\, \mathbf{out}(\mathit{from})@\mathit{PrintServer}.\mathbf{nil} \\
\\
\mathit{PClient} :: \mathbf{rec}X.\mathbf{in}(\text{“}\mathit{PrintSlot}\text{”})@\mathit{PrintServer}. \\
\qquad\qquad \mathbf{out}(\text{“}\mathit{Print}\text{”}, \mathit{PClient})@\mathit{PrintServer}. \\
\qquad\qquad \mathbf{in}(\text{“}\mathit{PrintOk}\text{”})@\mathit{PClient}.X
\end{array}
\qquad
\begin{array}{l}
\mathit{PrintServer} :: \langle \text{“}\mathit{PrintSlot}\text{”} \rangle \\
\qquad\qquad\quad | \ \langle \text{“}\mathit{PrintSlot}\text{”} \rangle \\
\qquad\qquad\quad | \ \mathbf{rec}X.\mathbf{in}(\text{“}\mathit{Print}\text{”}, !\,\mathit{from})@\mathit{PrintServer}. \\
\qquad\qquad\qquad (X \,|\, \mathbf{out}(\mathit{from}, \mathit{docname})@\mathit{Printer}. \\
\qquad\qquad\qquad\quad \mathbf{in}(\mathit{from})@\mathit{PrintServer}. \\
\qquad\qquad\qquad\quad \mathbf{out}(\text{“}\mathit{PrintOk}\text{”})@\mathit{from}. \\
\qquad\qquad\qquad\quad \mathbf{out}(\text{“}\mathit{PrintSlot}\text{”})@\mathit{PrintServer}.\mathbf{nil})
\end{array}
$$

Table 4. A simple μKLAIM system

a “*PrintSlot*” hence sends to the server the document to print and wait for the result. *PClient* is a possible μKLAIM implementation for a node.

3 A Modal Logic for μKLAIM

For specifying dynamic properties of μKLAIM systems, in the proposed framework the classical diamond operator ($\langle\cdot\rangle$) is indexed with *label predicates*: A net N satisfies a formula $\langle\mathcal{A}\rangle\phi$, w.r.t. a given set of localities L, if there exists a label λ and a net N' such that we have: $N \succ\!\xrightarrow{\lambda}_L N'$, λ satisfies $\mathcal{A}$ and N' satisfies ϕ. Process predicates are used to specify *static* properties of processes that are spawned to be evaluated remotely. These predicates shall permit specifying properties about the accesses to the resources of the net (data and nodes) that a process might perform in a computation and their causal dependencies. The logic provides also state formulae for specifying the distribution of resources (i.e. data stored in nodes) in the system. The logic has been equipped with a sound and partially complete proof system (not presented here) based on tableaux and inspired by [6, 15, 17].

Let Φ be the set of logic formulae defined by the grammar of Table 5, where ϕ denotes logical formulae, κ is logical variable belonging to the set *VLog*, $\mathcal{A}$ denotes a *label predicate*, i.e. a predicate that finitely specifies an infinite set of transition labels, and pp denotes a *process predicate* that express *static* properties of processes. Finally, $\widetilde{u:d}$ denotes a finite sequence of pairs $u_i : d_i$ where u_i is a locality variable while d_i is a term denoting a subset of localitites.

A formula ϕ can be either **true**, and this is satisfied by every net, or a composite formula: A net N satisfies $\phi_1 \vee \phi_2$ if N satisfies either ϕ_1 or ϕ_2, while N satisfies $\neg\phi$ if N does not satisfies ϕ. Specific state formulae ($\langle t\rangle@\ell$) are introduced for specifying properties related to the data placement over the nodes. N satisfies $\langle t\rangle@\ell$ if and only if N contains node ℓ and tuple $\langle\mathcal{T}[\![t]\!]\rangle$ is stored in the tuple space located at ℓ. Dynamic properties of μKLAIM systems are specified using the operator *diamond* ($\langle\mathbf{E}\,\widetilde{u:d}.\mathcal{A}\rangle\phi$) that is indexed with predicates specifying properties of transition labels. Function $\mathbb{A}[\![\cdot]\!]$ interprets each predicate $\mathcal{A}$ as a set of transition labels while $\Sigma(\cdot)$ interprets a sequence $\widetilde{u:d}$ as a set of substitution. Let L be a set of localities, the intuitive interpretation of $\langle\mathbf{E}\,\widetilde{u:d}.\mathcal{A}\rangle\phi$ will be: A net N satisfies $\langle\mathcal{A}\rangle\phi$ if there exist σ, λ and N' such that: $\sigma \in \Sigma(\widetilde{u:d})$, $\langle \in \mathbb{A}[\![\mathcal{A}\sigma]\!]$, $N \succ\!\xrightarrow{\lambda}_L N'$ and N' satisfies $\phi\{\sigma\}$. Functions $\Sigma(\cdot)$ and $\mathbb{A}[\![\cdot]\!]$ will be formally defined later (see Table 6 and Table 7). We omit $\mathbf{E}\,\widetilde{u:d}$ when $\widetilde{u:d}$ is empty.

$$
\begin{array}{rcl}
\Phi & ::= & \mathbf{true} \mid \langle t\rangle@\ell \mid \langle\mathbf{E}\,\widetilde{u:d}.\mathcal{A}\rangle\phi \mid \kappa \mid \nu\kappa.\phi \mid \phi\vee\phi \mid \neg\phi \\
d & ::= & \star \mid \{\widetilde{l}\} \mid d_1 - d_2 \\
\mathcal{A} & ::= & \circ \mid \ell_1 : \alpha@\ell_2 \mid \mathcal{A}_1 \cap \mathcal{A}_2 \mid \mathcal{A}_1 \cup \mathcal{A}_2 \\
\alpha & ::= & \mathtt{O}(t) \mid \mathtt{I}(t) \mid \mathtt{R}(t) \mid \mathtt{E}(\mathtt{pp}) \\
\mathtt{pp} & ::= & \mathtt{1_P} \mid \mathtt{ap} \rightarrow \mathtt{pp} \mid \mathtt{pp} \wedge \mathtt{pp} \\
\mathtt{ap} & ::= & \mathtt{o}(t)@\mathtt{lp} \mid \mathtt{i}(T)@\mathtt{lp} \mid \mathtt{r}(T)@\mathtt{lp} \mid \mathtt{e}(\mathtt{pp})@\mathtt{lp}
\end{array}
$$

Table 5. The logic for μKLAIM

Recursive formulae $\nu\kappa.\phi$ are used to specify *infinite* properties of systems. 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.

Other formulae like $[\mathbf{E}\,\widetilde{u:d}.\mathcal{A}]\phi$, $\phi_1 \wedge \phi_2$ or $\mu\kappa.\phi$ can be expressed in ϕ. Indeed $[\mathbf{E}\,\widetilde{u:d}.\mathcal{A}]\phi = \neg\langle\mathbf{E}\,\widetilde{u:d}.\mathcal{A}\rangle\neg\phi$, $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$ and $\mu\kappa.\phi = \neg\nu\kappa.\neg\phi[\neg\kappa/\kappa]$. We shall use these derivable formulae as *macros* in ϕ.

$$\mathbb{M}^L[\![\mathbf{true}]\!]\varepsilon = Net \qquad \mathbb{M}^L[\![\phi_1 \vee \phi_2]\!]\varepsilon = \mathbb{M}^L[\![\phi_1]\!]\varepsilon \cup \mathbb{M}^L[\![\phi_2]\!]\varepsilon$$
$$\mathbb{M}^L[\![\kappa]\!]\varepsilon\sigma = \varepsilon(\kappa) \qquad \mathbb{M}^L[\![\neg\phi]\!]\varepsilon = Net - \mathbb{M}^L[\![\phi]\!]\varepsilon$$
$$\mathbb{M}^L[\![\langle t\rangle@l]\!]\varepsilon = \{N | \exists N_1 : N \equiv N_1 \parallel l :: \langle \mathcal{T}[\![t]\!]\sigma\rangle\} \quad \mathbb{M}^L[\![\nu\kappa.\phi]\!]\varepsilon = \bigcup\{\mathcal{N} | \mathcal{N} \subseteq \mathbb{M}^L[\![\phi]\!]\varepsilon \cdot [\kappa \mapsto \mathcal{N}]\}$$
$$\mathbb{M}^L[\![\langle \mathbf{E}\ \widetilde{u : d}.\mathcal{A}\rangle\phi]\!]\varepsilon = \{N | \exists\sigma \in \Sigma(\widetilde{u : d}) \exists\lambda \in \mathbb{A}[\![\mathcal{A}\sigma]\!] \exists N'.N \overset{\lambda}{\rightarrowtail}_L N', N' \in \mathbb{M}^L[\![\phi\sigma]\!]\}$$
$$\Sigma(\widetilde{u : d}) = \{\sigma | \forall u_i : d_i \in \widetilde{u : d}.\sigma(u) \in \mathbb{D}[\![d_i]\!]\}$$

Table 6. Formulae interpretation function

$$\mathbb{A}[\![\circ]\!] = \Lambda$$
$$\mathbb{A}[\![l_1 : \mathtt{O}(et)@l_2]\!] = \{l_1 : \mathbf{o}(et)@l_2\} \quad \mathbb{A}[\![l_1 : \mathtt{I}(et)@l_2]\!] = \{l_1 : \mathbf{i}(et)@l_2\}$$
$$\mathbb{A}[\![l_1 : \mathtt{R}(et)@l_2]\!] = \{l_1 : \mathbf{r}(et)@l_2\} \quad \mathbb{A}[\![l_1 : \mathtt{E}(et)@l_2]\!] = \{l_1 : \mathbf{e}(et)@l_2\}$$
$$\mathbb{A}[\![\mathcal{A}_1 \cap \mathcal{A}_1]\!] = \mathbb{A}[\![\mathcal{A}_1]\!] \cap \mathbb{A}[\![\mathcal{A}_2]\!] \qquad \mathbb{A}[\![\mathcal{A}_1 \cup \mathcal{A}_1]\!] = \mathbb{A}[\![\mathcal{A}_1]\!] \cup \mathbb{A}[\![\mathcal{A}_2]\!]$$

Table 7. Label Predicates Interpretation

The interpretation function of formulae makes use of *logical environments*, functions that, given a logical variable, yield a set of nets.

Let *VLog* be the set of logical variables and *Net* be the set of μKLAIM nets, we define the *logical environment Env* as a subset of $[VLog \to 2^{Net}]$. We should use ε, sometime with indexes, to denote elements of *Env*.

Let L be a set of localities, the interpretation function $\mathbb{M}^L[\![\cdot]\!] : \Phi \to Env \to 2^{Net}$ that, using a logical environment, for each $\phi \in \Phi$, yields the set of nets that satisfy ϕ or, equivalently, the set of nets that are *models* for ϕ with respect to a logical environment. Function $\mathbb{M}^L[\![\cdot]\!]$ is formally defined in Table 6 where is used function $\mathbb{D}[\![\cdot]\!]$ defined as follows:

$$\mathbb{D}[\![\star]\!] = \mathcal{L}\mathbb{D}[\![\{l_1, \ldots, l_n\}]\!] = \{l_1, \ldots, l_n\}\mathbb{D}[\![d_1 - d_2]\!] = \mathbb{D}[\![d_1]\!] - \mathbb{D}[\![d_2]\!]$$

Please notice that the logic interpretation function is parametrized with respect to a given set L of localities: formulae will state only properties regarding these set of names. Standard logic interpretation can be obtained considering $L = \mathcal{L}$, i.e. the set of all localities.

We will write $N \models_L \phi$ if $N \in \mathbb{M}^L[\![\phi]\!]\varepsilon_0$ where ε_0 is used to denote the empty logical environment.

A label predicate $\mathcal{A}$ is built from $\circ$ and *abstract actions* ($\ell_1 : \alpha@\ell_2$) by using disjunction ($\cdot \cup \cdot$) and conjunction ($\cdot \cap \cdot$).

Predicate $\circ$ is used to denote the set of all transition labels. *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 *process predicates* instead of processes.

Formal interpretation of label predicates is defined by means of interpretation function $\mathbb{A}[\![\cdot]\!]$. This function takes a label predicate $\mathcal{A}$ and yields a set of transition labels. Some clausules that defines $\mathbb{A}[\![\cdot]\!]$ can be found in Table 7.

A property like: *never a process, which is not located at l_1, retrieves the tuple et from the tuple space located at l_2* would be rendered as: $\nu\kappa.\neg\langle \mathbf{E}\ u : (\star - l_1).u : \mathtt{I}(et)@l_2\rangle\mathbf{true} \wedge [\circ]\kappa$.

Process predicates shall be used to specify the kind of accesses to the resources of the net (data and nodes) that a process might perform in a computation. These accesses are composed for specifying their causal dependencies. The The properties we would like to express are of the form *"first read something and then use the acquired information in some way"*.

We use $\mathtt{1_P}$ for a generic process and $\mathtt{pp}_1 \wedge \mathtt{pp}_2$ for the set of processes that *satisfy* $\mathtt{pp}_1$ and $\mathtt{pp}_2$. A process satisfies $\mathtt{ap} \to \mathtt{pp}$ if it may perform an access (i.e. an action) that satisfies ap and use the acquired information as specified by pp. The satisfaction relation between actions (act) and access predicates (ap) is quite intuitive. For instance, in the case of **out** and **eval** we have that $\mathbf{out}(t)@\ell_2$ satisfies $\mathtt{o}(t)@\ell_2$ while $\mathbf{eval}(P)@\ell$ satisfies $\mathtt{e}(\mathtt{pp})@\ell$ if and only if P satisfies pp.

Process predicates can be thought of 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. For

$$\mathbb{P}[\![\mathtt{1_P}]\!] = \mathit{Proc} \qquad \mathbb{P}[\![\mathtt{pp}_1 \wedge \mathtt{pp}_2]\!] = \mathbb{P}[\![\mathtt{pp}_1]\!] \cap \mathbb{P}[\![\mathtt{pp}_2]\!]$$

$$\mathbb{P}[\![\mathtt{ap} \to \mathtt{pp}]\!] = \{P \mid \exists \mathit{act}, Q_1, Q_2 : P \equiv_\alpha Q_1, Q_1 \xrightarrow[\mathtt{fv}(\mathtt{ap} \to \mathtt{pp})]{\mathit{act}} Q_2, \mathit{act} \in \mathbb{AC}[\![\mathtt{ap}]\!], Q_2 \in \mathbb{P}[\![\mathtt{pp}]\!]\}$$

where $\mathbb{AC}[\![\cdot]\!]$ is defined as follows:

$$\mathbb{AC}[\![\mathtt{o}(t)@\ell]\!] = \{\mathbf{out}(t)@\ell\} \qquad \mathbb{AC}[\![\mathtt{i}(T)@\ell]\!] = \{\mathbf{in}(T)@\ell\}$$

$$\mathbb{AC}[\![\mathtt{r}(T)@\ell]\!] = \{\mathbf{read}(T)@\ell\} \qquad \mathbb{AC}[\![\mathtt{e}(\mathtt{pp})@\ell]\!] = \{\mathbf{eval}(Q)@\ell \mid Q \in \mathbb{P}[\![\mathtt{pp}]\!]\}$$

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

Table 9. Process predicates interpretation functions

these reasons process predicates are not able to distinguish statically $P_1 + P_2$ from $P_1|P_2$. These processes will be discriminated by the full logic when that are executed on a node.

To formally define function $\mathbb{P}[\![\cdot]\!]$ that yields the set of process satisfying a given process predicates, we need to introduce a transition relation for describing possible computations of processes. The operational semantics proposed in Table 3, is not adequate, because it describes the actual computation of nets and processes. Instead, we need an a sort of *abstract interpretation* of processes that models the structured sequences of actions a process might perform during its computation.

Let $\mathcal{V}$ be a set of variables, we will write $P \xRightarrow[\mathcal{V}]{\mathit{act}} Q$ if: the process P, at some point of its computation, can perform action *act* and all the actions, that syntactically precede *act*, executed before *act*, do not bind variables in $\mathcal{V}$.

Definition 2. *Let $P \to_{\mathcal{V}} Q$ be the relation defined in Table 8, $P \xRightarrow[\mathcal{V}]{\mathit{act}} Q$ is inductively defined as follows:*

- *for every $\mathcal{V}$, $\mathit{act}.P \xRightarrow[\mathcal{V}]{\mathit{act}} P$*
- *if $P \to_{\mathcal{V}} P'$ and $P' \xRightarrow[\mathcal{V}]{\mathit{act}} Q$ then $P \xRightarrow[\mathcal{V}]{\mathit{act}} Q$*

$$\frac{bv(\mathit{act}) \cap \mathcal{V} = \emptyset}{\mathit{act}.P \to_{\mathcal{V}} P} \qquad \mathbf{rec}X.P \to_{\mathcal{V}} P[\mathbf{rec}X.P/X]$$

$$P|Q \to_{\mathcal{V}} P \qquad P + Q \to_{\mathcal{V}} P$$

Table 8. Abstract interpretation of processes (simmetric rules for $|$ and $+$ are omitted)

Process predicates interpretation function $\mathbb{P}[\![\cdot]\!]$ is inductively defined in Table 9. We will write $P : \mathtt{pp}$ to denote that $P \in \mathbb{P}[\![\mathtt{pp}]\!]$. Conversely, we will write $\neg(P : \mathtt{pp})$ whenever $P \notin \mathbb{P}[\![\mathtt{pp}]\!]$.

We would like to remark that process predicates represent *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.

We can now turn our attention to an interesting subset of our logic, namely the one including only *positive formulae*. We define Φ^+ as the set of formulae ϕ defined using the following grammar:

$$\phi ::= \mathbf{true} \mid \mathbf{false} \mid t@\ell \mid \langle \mathcal{A} \rangle \phi \mid \kappa \mid \nu\kappa.\phi \mid \mu\kappa.\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

Positive formulae can be used for specifying liveness properties. These are useful when one has to establish *access right violations*, i.e. properties like *"a given sequence of transition can be executed in the system"*. We will use positive formulae later when we consider refinement of context specifications.

Example 2. Let us consider some interesting properties for the print server described in Example 1. The first property we can specify is that the system is deadlock-free: $\nu\kappa.\langle\circ\rangle\kappa \wedge [\circ]\kappa$

Another interesting propery shold be that never a process that is not located at *PrintServer* can directly send documents to the *Printer*: $\neg(\mu\kappa.\langle \mathbf{E}\, u_1 : (\star - \{\mathit{PrintServer}\}).u_2 : \star. : u_1 : o(u_2)@\mathit{Printer}\rangle \mathbf{true} \vee \langle\circ\rangle\kappa)$

Please notice that, the second formula relies on the negation of a positive formula.

4 Context specifications

In order to establish properties of network oriented applications it is definitively too demanding to require full knowledge of the whole system. Often, only part of the application is known and the knowledge of the rest is limited to the possible interactions of the environment with the known part and to the resources made available by the environment. Here, we propose an approach that permits representing the known part of the system by means of a *concrete language*, μKLAIM in our case. For representing the *unknown* part, instead, we introduce instead an ad hoc formalism that enables us to provide an abstract description, that we call *context* of the environment.

Indeed, a network application can be thought as composed of two parts:

- the part that we completely know;
- a partial specification of the rest of the system that models context.

By following this idea, we define a framework for finitely specifying contexts for μKLAIM nets and introduce an operational semantics for μKLAIM net within a context. We shall also define a notion of *agreement* of a μKLAIM net N with a context specification $C[\ ]$ that will enable us to study the effect of specific concretions of the abstract specification.

The specification of the context shall take into account two aspects: the resources provided by the environment (sites and tuples) and the interactions that the environment is willing to engage with.

∇ is the set of *specification of context n* defined in Table 10, where the same notation of Section 2 is used. Terms $@\ell$ and $et@\ell$ indicate that within the context there is a node whose name is ℓ and that tuple *et* is available at node ℓ.

$\exists u.n$ plays the role of existential quantification; it states that if l is a *new/fresh* locality then context garantees existence of a component that behaves like $n[l/u]$. Operator ! is used to model contexts containing an unbouded number of components each of wich specified by n. Term $!n$ can be read as $n \otimes !n$.

Computational specification of the context is described by terms of the form $\ell_1 : act.p$, which states that at locality ℓ_1 there is a process that can perform action *act* and that the remaining context is p.

Context specifications are composed via $\otimes$ and $\oplus$: $n_1 \otimes n_2$ states that the context behaves like both n_1 and n_2 while $n_1 \oplus n_2$ states that one of n_1 and n_2 specifies the behaviour of the context. Term $\mathbf{rec}\pi.p$ is used for *recursive contexts* where it is assumed that all occurrences of π in p are guarded.

$$
\begin{array}{l}
n ::= p \mid \exists u.n \mid n_1 \otimes n_2 \mid n_1 \oplus n_2 \mid\ !n \\
p ::= \mathbf{0} \mid (l : act).p \mid t@\sigma \mid \rho@\sigma \\
\qquad \mid \mathbf{rec}\pi.p \mid \pi \mid p \otimes p \mid p \oplus p
\end{array}
$$

Table 10. Net Contexts

From now on we will consider *Abstract Nets*, i.e. μKLAIM nets that work in a context ($n\,[N]$). The operational semantics of abstract nets is presented in Table 3.

Example 3. The following is a possible specification for a client running in the context of the print server presented in Example 1:

$$\exists u.\mathbf{rec}\pi.u : \mathbf{in}(\text{“}PrintSlot\text{”})@PrintServer.u : \mathbf{out}(u)@PrintServer.u : \mathbf{in}(\text{“}PrintOk\text{”})@u.\pi$$

In the next section, we will show that the *PClient* presented in Table 4 *agrees* with the above specification.

5 Context refinement and property preservation

By using the operational semantics presented in the previous section, we are able to consider the computations of a net within a partially specified context. By exploiting the operational semantics of context it is then possible to interpret the modal logic formulae of Section 3 also over *contextualized systems*. To export the properties established for abstract nets to *concrete nets* (i.e. systems where we have replaced the context specification with a *concrete* implementation) we need a notion of *refinement* between concrete implementations and context specifications that guarantees preservation of formulae satisfaction.

$$\frac{\mathcal{T}[\![t]\!] = et}{n \otimes l_1 : \mathbf{out}(t)@l_2.p\,[N \parallel l_2 :: Q] \succ\!\!\xrightarrow{l:\mathbf{o}(et)@l'} n \otimes p\,[N \parallel l_2 :: Q \parallel l_2 :: \langle et \rangle]}$$

$$n \otimes l_1 : \mathbf{eval}(Q)@l'.p\,[N \parallel l_2 :: P] \parallel\!\succ\!\!\xrightarrow{l_1:\mathbf{e}(Q)@l_2} n \otimes p\,[N \parallel l_2 :: P_2|Q]$$

$$\frac{match(\mathcal{T}[\![T]\!], et) = \sigma}{n \otimes l_1 : \mathbf{in}(l)@T_2.p\,[N \parallel l_2 :: \langle et \rangle] \succ\!\!\xrightarrow{l_1:\mathbf{i}(et)@l_2} n \otimes p\,[N \parallel l_2 :: \mathbf{nil}]}$$

$$\frac{match(\mathcal{T}[\![T]\!], et) = \sigma}{n \otimes l_1 :: \mathbf{read}(T)@l_2.p\,[N \parallel l_2 :: \langle et \rangle] \succ\!\!\xrightarrow{\mathbf{r}(l_1)etl_2} n \otimes p\,[N \parallel l_2 :: \langle et \rangle]}$$

$$\frac{l \notin n \otimes p\,[N] \quad n \otimes p[l/u]\,[N] \succ\!\!\xrightarrow{\lambda} n'\,[N']}{n \otimes \exists u.p\,[N] \succ\!\!\xrightarrow{\lambda} n'\,[N']} \qquad \frac{n \otimes n_1\,[N_1] \succ\!\!\xrightarrow{\lambda} n \otimes n_2\,[N_2]}{n \otimes !n_1\,[N] \succ\!\!\xrightarrow{\lambda} n \otimes n_2 \otimes !n_1\,[N_1]}$$

$$\frac{n_2\,[N_1 \parallel l :: \mathbf{nil}] \succ\!\!\xrightarrow{\lambda} n_2\,[N_2]}{n \otimes @l\,[N_1] \succ\!\!\xrightarrow{\lambda} n_2\,[N_2]} \qquad \frac{n_2\,[N_1 \parallel l :: \langle et \rangle] \succ\!\!\xrightarrow{\lambda} n_2\,[N_2]}{n \otimes \langle et \rangle@l\,[N_1] \succ\!\!\xrightarrow{\lambda} n_2\,[N_2]}$$

Table 11. Context Labelled Operational Semantics

Below, we shall define such a notion and shall show that if a net N *is a refinement* of a context specification n, i.e, the latter is partially instantiated (implemented) as a concrete net, then all formulae satisfied by the more abstract description are satisfied also by the refined one.

More specifically, we have that if $n \otimes n_1\,[N]$ satisfies a formula ϕ and n_1 refines N_1 then also $n\,[N \parallel N_1]$ satisfies ϕ. The context refinement is defined using two relations between μKLAIM nets ($\sqsubseteq_L^+$ and $\simeq_L$), which will be formally defined later in this section and capture the intuition that smaller net can exhibit fewer behaviours, where these are used to describe the set of all possible computations of a net.

Definition 3. *A net* N approximates *the context* $\exists u_1 \ldots \exists u_n.p$ *within the abstract net* $n_1\,[N_1]$ *if and only if there exist* $l_1, \ldots, l_n \in N$ *such that:* $n_1 \otimes p[l_1/u_1, \ldots, l_n/u_n]\,[N_1] \sqsubseteq^+_{loc(N_1)} n_1\,[N_1 \parallel N]$

Definition 4. *A net* N agrees *with the context* $\exists u_1 \ldots \exists u_n.p$ *within the abstract net* $n_1\,[N_1]$ *if and only if there exists* $l_1, \ldots, l_n \in N$ *such that:* $n_1 \otimes p[l_1/u_1, \ldots, l_n/u_n]\,[N_1] \simeq_{loc(N_1)} n_1\,[N_1 \parallel N]$

In the rest of the section we shall prove that:

if a net N *approximates* p over $n_1\,[N_1]$ then for each positive formula ϕ
$n_1 \otimes p\,[N_1] \models_L \neg\phi$ then we also have $n_1\,[N \parallel N_1] \models_L \neg\phi$.

This permits guaranteeing that un-satisfaction of positive formulae is preserved by context aproximation. Since positive formulae can be used for expressing *violation of accesses*, this means that if N aproximates p with respect to $n_1\,[N_1]$ and $n_1 \otimes p\,[N_1]$ does not violate given *access rights*, these are not violated also by $n_1\,[N \parallel N_1]$.

The other result that we shall establish connects formulae satisfaction to context agreement: If a context specification is partially instantiated (implemented) as a concrete net then all formulae satisfied by the more abstract description are satisfied also by the refined one:

If N *agrees* with p over $n_1\,[N_1]$ then each formula satisfied by $n_1 \otimes p\,[N_1]$ is also satisfied by
$n_1\,[N \parallel N_1] \models \phi$.

We shall now introduce a behavioural equivalence between μKLAIM nets that is based on a preorder over computation trees of nets; we shall call *net behaviours* such computation trees. To define the equivalence we need to introduce two preorders for comparing processes and transition labels.

The preorder for processes is defined by relying on the transition $\stackrel{act}{\Longrightarrow}_{\mathcal{V}}$ introduced in Definition 2.

Definition 5. *Relation $\trianglerighteq^i$ is inductively defined as follows:*

- *for every process P and Q, $P \trianglerighteq^0 Q$;*
- *$P \trianglerighteq^{i+1} Q$ if and only if:*
  - *if $Q \stackrel{act}{\Longrightarrow}_{\mathcal{V}} Q'$ and $act \neq \mathbf{eval}(Q_1)@l$ then $P \stackrel{act}{\Longrightarrow}_{\mathcal{V}} P'$ and $P' \trianglerighteq^i Q'$;*
  - *if $Q \stackrel{\mathbf{eval}(Q_1)@l}{\Longrightarrow}_{\mathcal{V}} Q'$ then $P \stackrel{\mathbf{eval}(P_1)@l}{\Longrightarrow}_{\mathcal{V}} P'$ and $P' \trianglerighteq^i Q'$, $P_1 \trianglerighteq^i Q_1$.*

Finally, $P \trianglerighteq Q$ if and only if there exists P' and Q' such that $P =_\alpha P'$, $Q =_\alpha Q'$ and $P' \trianglerighteq^i Q'$ for every i.

Preorder $\trianglerighteq$ guarantees that, if $P \trianglerighteq Q$ then P performs all *accesses* of Q. In other words, every sequence of actions performed by Q can be also be performed by P; however the sequences of P can be intermixed with other actions.

The first result we have is that $P \trianglerighteq Q$ implies that P satisfies all process predicates satisfied by Q.

Theorem 1. *If $P \trianglerighteq Q$ and $Q : \mathcal{pp}$ then $P : \mathcal{pp}$.*

Labels in μKLAIM are complex and also for them a preorder is needed for abstracting from some details. This relation is defined by taking advantage of the equivalence between processes introduced above.

Definition 6. *Let $\preceq$ be the smallest relation that satisfies:*

- $\lambda \preceq \lambda$
- $l_1 : \mathbf{e}(P)@l_2 \preceq l_1 : \mathbf{e}(Q)@l_2$ *if* $P \trianglerighteq Q$.

Intuitively $\lambda_1 \preceq \lambda_2$ if and only if they have the same structure and the corresponding mobile components are related via $\trianglerighteq$.

Theorem 2. *If $\lambda_1 \preceq \lambda_2$ then for each label predicate $\mathcal{A}$, if $\lambda_2 \in \mathbb{A}[\![\mathcal{A}]\!]$ then $\lambda_1 \in \mathbb{A}[\![\mathcal{A}]\!]$.*

Nets will be compared according to their computational trees or *behaviours*, these are built according to the following syntax, where λ is a transition label from Λ as defined in Section 2:

$$\Gamma ::= \bot \;\mid\; \top \;\mid\; et@\ell \;\mid\; \overline{et@\ell}\,\lambda \to \Gamma \;\mid\; \overline{\lambda} \;\mid\; \Gamma \wedge \Gamma \;\mid\; \omega$$

The set of all possible behaviours will be denoted by $\mathcal{B}$, while the set of *positive behaviours*, i.e. of behaviours that do not contain neither $\overline{\lambda}$ nor $\overline{et@l}$ as sub-term, is denoted by $\mathcal{B}^+$.

Let L be a set of localities, we will write $N \vdash_L \Gamma$ to indicate that net N has behavior Γ. The symbol $\bot$ is used to represent *fully unspecified behaviour* and every net N is assumed to have $\bot$ as a possible behaviour ($N \vdash_L \bot$ holds for any N and L). We have $N \vdash_L et@l$ if and only if $l \in L$, $l \in N$ and et is in the tuple spaces at l.

The behaviour $\lambda \to \Gamma$ represents the set of those systems that are able to perform an action equivalent to λ and have behaviour Γ. Thus,we have $N \vdash_L \lambda \to \Gamma'$ if and only if there exist N' such that $N \succ\!\!\stackrel{\lambda}{\longrightarrow}_L N'$ and $N' : \Gamma'$. Viceversa a net has a behaviour $\overline{\lambda}$ if it cannot evolve with a transition labelled λ. Formally, $N \vdash \overline{\lambda}$ if and only if there is no N' such that $N \succ\!\!\stackrel{\lambda}{\longrightarrow}_L N'$.

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 $N_2 : \Gamma_2$).

Behaviour ω represents the capability of performing any actions; no net has behaviour ω. It is used for inconsistent behaviours: ω is equivalent to behaviours like $\overline{\lambda} \wedge \lambda \to \bot$ that cannot be satisfied by any net.

Behaviours will be compared via an ordering relation, $\leq$, that is defined in a such way that $\bot$ is the minimum and ω the maximum.

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

- $\omega \leq \Gamma$
- $\Gamma \leq \bot$
- $\lambda_1 \to \Gamma_1 \leq \lambda_2 \to \Gamma_2$ *if* $\Gamma_1 \leq \Gamma_2$, *and* $\lambda_1 \preceq \lambda_2$
- $\Gamma_1 \wedge \Gamma_2 \leq \Gamma_2 \wedge \Gamma_1$
- $\Gamma_1 \wedge \Gamma \leq \Gamma_1$
- $\Gamma_1 \wedge \Gamma \leq \Gamma_2 \wedge \Gamma$ *if* $\Gamma_1 \leq \Gamma_2$

If we interpret behaviours as requirements over the actions executed by a net during its computations then the ordering $\Gamma_1 \leq \Gamma_2$ indicates that a net with behaviour Γ_1 satisfies more requirements than a net with a behaviour Γ_2.

Definition 8. *Let L be a set of localities and let* N_1 *and* N_2 *be nets that do not leave L forever (see Definition 1):*

1. $N_1 \sqsubseteq_L^+ N_2$ *if and only if for all* $\Gamma_2 \in \mathcal{B}^+$ *such that* $N_2 \vdash_L \Gamma_2$ *there exists* $\Gamma_1 \in \mathcal{B}^+$ *such that* $N_1 \vdash_L \Gamma_1$ *and* $\Gamma_1 \leq \Gamma_2$.
2. $N_1 \sqsubseteq_L N_2$ *if and only if for all* $\Gamma_2 \in \mathcal{B}$ *such that* $N_2 \vdash_L \Gamma_2$ *then there exists* $\Gamma_1 \in \mathcal{B}$, *with* $N_1 \vdash_L \Gamma_1$, *such that* $\Gamma_1 \leq \Gamma_2$;
3. $N_1 \simeq_L N_2$ *if and only if* $N_1 \sqsubseteq_L N_2$ *and* $N_2 \sqsubseteq_L N_1$.

Theorem 3. *Let L be a set of localities, for each* N_1, N_2 *and* ϕ, *If* $N_1 \simeq N_2$ *and* $N_1 \models_L \phi$ *then* $N_2 \models_L \phi$.

Theorem 4. *Let L be a set of localities, for each* N_1, N_2 *and* $\phi \in \mathcal{L}^+$, $N_1 \sqsubseteq_L^+ N_2$, $N_2 \models \phi$ *then* $N_1 \models \phi$.

Theorem 5. *If N* approximates *p over* $n_1\,[N_1]$, *for each* $\phi \in \mathcal{L}^+$, *if* $n_1 \otimes p\,[N_1] \models_L \neg\phi$ *then* $n_1\,[N \parallel N_1] \models \neg\phi$

Theorem 6. *If N* agrees *with p over* $n_1\,[N_1]$, *for each* $\phi \in \mathcal{L}_L$, $n_1 \otimes p\,[N_1] \models \phi$ *if and only if* $n_1\,[N \parallel N_1] \models \phi$

Example 4. Please notice that the node *PClient* of Table 4 agrees the context of Example 3, within the net obtained by parallel composition of *PrintServer* and *Printer* of Table 4. This means the satisfaction of formulae of Example 2 is preserved. Another possible implementation for the print client should be the following:

$$\begin{array}{c} \mathit{PProxy} :: \mathbf{rec}X.\mathbf{in}(!u)@\mathit{PProxy}.\mathbf{out}(\mathit{PProxy})@\mathit{PrintServer}. \\ \mathbf{in}(\text{“}\mathit{PrintOk}\text{”})@\mathit{PProxy}.\mathbf{out}(\text{“}\mathit{PrintOk}\text{”})@u.X \parallel \\ \mathit{PClient}2 :: \mathbf{rec}X.\mathbf{out}(\mathit{PClient}2)@\mathit{PProxy}.\mathbf{in}(\text{“}\mathit{PrintOK}\text{”})@\mathit{PClient}2.X \end{array}$$

Also this implementation *agrees* the proposed context specification.

6 Conclusions

We have introduced a framework for specifying contexts for KLAIM nets. By means of contexts, we are able to provide abstract specifications of a given system and avoid describing all of its components in full. Indeed, some of these components could be known or implemented only at a later stage. Then, the implemented component can be removed from the context and added to the implemented part thus performing a *concretion* operation.

A context over a net N can be viewed as the composistion of a given set of specifications. One can obtain a full implementated system by progressively replacing specifications with concrete implementations. The proposed framework guarantees that at each stage of refinement, if the introduced implementation agrees with the specification, satisfaction of formulae will be preserved. Indeed, for specific properties, definable via positive formulae, we can guarantee that if a net N in a specific context $C[\,]$ satisfies a given formula then every net obtained by composing N with a context, that can be considered a concretion of $C[\,]$ does the same.

The framework can be seen as a starting point for defining a methodology for developing and analyzing network applications. Context specification can also be used for modelling *intruders*, i.e. malicious components that try to modify the behaviour of a systems. Properties established for nets with intruders can then be useful for guaranteeing security requirements.

References

1. L. Bettini, V. Bono, , R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. submitted, 2003.
2. L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. sv, 1998.
3. L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *27th Annual Symposium on Principles of Programming Languages (POPL) (Boston, MA)*. ACM, 2000.
4. L. Cardelli and A. D. Gordon. Logical properties of name restriction. In *Proceeding of International Conference on Typed Lambda Calculi and Applications, TLCA'01*, volume 2044 of *Lecture Notes in Computer Science*. Springer, 2001.
5. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(10):444–458, October 1989. Technical Correspondence.
6. R. Cleaveland. Tableau-based model checking in the propositional μ-calculus. *Acta Informatica*, 27(8):725–747, September 1990.
7. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
8. R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.
9. R. De Nicola and M. Loreti. Open nets, contexts and their properties (full paper). Avaiable at `ftp://music.dsi.unifi.it/papers/`.
10. R. De Nicola and M. Loreti. A Modal Logic for Mobile Agents. *ACM Transactions on Computational Logic*, 5(1), 2004.
11. C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.
12. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
13. D. Gelernter. Multiple Tuple Spaces in Linda. In J. G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *Lecture Notes in Computer Science*, pages 20–27, 1989.
14. M. Hennessy and J. Riely. Distributed processes and location failures. *Theoretical Computer Science*, 266(1–2):693–735, Sept. 2001.
15. C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, Oct. 1991.
16. J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science. Springer, 1999.
17. G. Winskel. A note on model checking the modal ν-calculus. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 761–772, Berlin, July 1989. Springer.