Specifying Properties for Modular π-Calculus*

Takashi KITAMURA Huimin LIN
Laboratory for Computer Science, Institute of Software, Chinese Academy of Sciences
{takashi, lhm}@ios.ac.cn

Abstract

We propose a modal logic for Modular πcalculus [12]: a logic to specify both temporal and spatial properties for processes in Modular πcalculus. Characterization of process equivalence the logic induce is investigated,and it is shown that the distinguishing power of the logic falls between bisimilarity and structural congruence. Then a model checking algorithm for the logic over the finite-control subset of Modular πcalculus is presented, and its correctness proved.

1. Introduction

Modular π-calculus ($M\pi$) [12] is a variant of the distributed π-calculus, Dπ [8]. $M\pi$ was designed to control *process modularity* in a flat location space, in the sense that several related computing processes can be "glued" to form a *mobility unit* so that they can move together to a destination; after arriving there the unit can be "broken" to allow its component processes to continue parallel execution. The main feature of $M\pi$ can be illustrated with a simple cab system example (CP), where a cab (Cab) can catch a client (Cl) in $city_a$, takes him to his destination ($city_d$) and unloads him upon arrival:

$$\begin{array}{ll} CP = & city_a\lfloor call(s).\mathbf{merge}.\overline{run}.\mathbf{go}\ s.\mathbf{split}.\overline{bye}.Cab\rfloor \\ & \parallel city_a\lfloor\overline{call}\langle city_d\rangle.bye.\overline{walk}.Cl\rfloor \\ \xrightarrow{\tau,city_a} & city_a\lfloor\mathbf{merge}.\overline{run}.\mathbf{go}\ city_d.\mathbf{split}.\overline{bye}.Cab\rfloor \parallel \\ & city_a\lfloor bye.\overline{walk}.Cl\rfloor \qquad (= E_1) \\ \xrightarrow{\tau,city_a} & city_a\lfloor\overline{run}.\mathbf{go}\ city_d.\mathbf{split}.\overline{bye}.Cab \mid bye.\overline{walk}.Cl\rfloor \\ & \qquad (= E_2) \\ \xRightarrow{\overline{run},city_a} & city_d\lfloor\mathbf{split}.\overline{bye}.Cab \mid bye.\overline{walk}.Cl\rfloor \qquad (= E_3) \\ \xrightarrow{\tau,city_d} & city_d\lfloor\overline{bye}.Cab\rfloor \parallel city_d\lfloor bye.\overline{walk}.Cl\rfloor \qquad (= E_4) \\ \xrightarrow{\tau,city_d} & city_d\lfloor Cab\rfloor \parallel city_d\lfloor\overline{walk}.Cl\rfloor \qquad (= E_5) \\ \xrightarrow{\overline{walk},city_d} & city_d\lfloor Cab\rfloor \parallel city_d\lfloor Cl\rfloor \qquad (= E_6) \end{array}$$

*Supported by the National Science Foundation of China (Grant No.60721061).

The system consists of two processes, a cab and a client, both located at $city_a$ initially. A mobility unit is syntactically surrounded with "$\lfloor$ "and "$\rfloor$". At the initial state the cab and client are in different units (as composed with "$\parallel$"). Modularity is controlled by two operators, "**merge**" and "**split**". In the example above, after the cab agrees with the client to take him to his destination city (via *call*), it joins the client process to form a single modularity unit, by executing **merge**. Now that the cab and the client are in the same modularity unit (as composed with "$\mid$"), cab can takes the client to $city_d$ by executing the command **go** $city_d$. After arriving at the city, the cab dissolves the unit, by executing **split**, so that the client can walk on his own. As demonstrated in the above example, transitions in $M\pi$ are of the form $M \xrightarrow{\alpha,l} M'$, labelled by an *action* α and a *location* l, meaning process M performs action α at location l evolving into M'.

The aim of this paper is to present a modal logic for $M\pi$; i.e., a logic to specify properties for systems in $M\pi$. It is designed that the logic is equipped with spatial as well as behavioural modalities. It is equipped with behavioural modalities to observe action behaviours of processes since they play a central role in specifying systems and also in theoretical discussion in $M\pi$. It is equipped with spatial modalities to observe spatial structures of processes, since spatial structure of processes in $M\pi$, such as distribution and modularity, is a key feature in $M\pi$ and hence it is useful to observe them to specify interesting properties in systems of Mπ.

To have a taste of the logic, the property "whenever there is a client in $city_a$ who wish to go to $city_d$, and there is a cab available in $city_a$, then the client will be able to arrive at $city_d$" can be described by the following formula:

$$(\langle\overline{call}\langle city_d\rangle@city_a\rangle\mathbf{T} \parallel \exists s.\langle call\ s@city_a\rangle\mathbf{T}) \Rightarrow \mathbf{EF}\langle\overline{walk}@city_d\rangle\mathbf{T}$$

Here $\parallel$ is a spatial operator: a process satisfying $A \parallel B$ if it is composed of two parallel agents, one satisfying A and the other satisfying B. In $\langle\alpha@l\rangle A$, where α is

an action and l is a location, $\langle \alpha @ l \rangle$ is a modal operator; a process M satisfies $\langle \alpha @ l \rangle A$ if it can perform a transition $M \xrightarrow{\alpha, l} M'$ and M' satisfies A. **T** is the constant proposition satisfied by any process, and **EF** is the CTL operator "eventually in the future" (which is a definable operator in our logic, see Section 3.3). More examples of the logic specifying characteristic properties of systems in $M\pi$ can be found in Section 3.4.

Our logic is a first-order extension of the propositional μ-calculus. Fixpoint formulas are constructed by applying the greatest and the least operators to *abstractions* which are (name-closed) functions from names to propositions. Thus fixpoint formulas are abstractions too. Abstractions can be applied to names to obtain propositions. In this way the difficulties concerning free names in recursive formulas can be avoided.

The first result of this paper is a characterization of the equivalence on processes induced by the logic. We show that the equivalence lies between structural congruence and bisimilarity in Section 4.

The second result concerns model checking problem of the logic. We show that the model checking problem for the *finite-control* fragment of $M\pi$ is decidable, by presenting a model checking algorithm along the lines of [19, 13]. We demonstrate that using the algorithm it can be automatically checked that the cab system CP, which is finite-control, actually satisfies the above formula.

Related work A number of efforts have been made to investigate modal logics for processes [1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19]. Among them, [1] studies a logic to observe behavioural and spatial properties for synchronous π-calculus. The design philosophy of the logic is similar to our logic, in the sense that the logic observes both behavioural and spatial properties of processes. But some characteristic design may be found in our logic especially for the part to involve spatial structure, reflecting the calculus design of $M\pi$. [3] provides a simple abstract model for distributed computing by extending CCS with process distribution over flat location space, and studies a logic for the model. Though the process model of [3] and $M\pi$ are similar, also some characteristic design are considered in our logic to suit for $M\pi$. Moreover, the logic in [3] has different design philosophy from our logic; i.e., the logic in [3] is designed with a few spatial constructs as "a simple spatial logic", while our logic is designed with common constructs of most other spatial logics. And these affect theoretical development, as discussed in Section 4.

2 Modular π-calculus

2.1 Syntax and semantics

We briefly present $M\pi$ here. The details of the calculus can be found in [12]. Assume an infinite set $\mathcal{N}_c$ of channel names, ranged over $a, b, c, \cdots$, and an infinite set $\mathcal{N}_l$ of location names, ranged over $l, k, \cdots$. We denote the union of $\mathcal{N}_c$ and $\mathcal{N}_l$ as $\mathcal{N}$, ranged over by $n, m \cdots$. The language of $M\pi$ is given by the following two-level syntax, which distinguish threads (P, Q) and agents (M, N):

$$\begin{array}{lll} \pi & ::= & \tau \mid a(n) \mid \bar{a}\langle n \rangle \mid \mathbf{go}\ l \mid \mathbf{merge} \mid \mathbf{split} \\ P, Q & ::= & 0 \mid \pi.P \mid [n = m]P \mid [n \neq m]P \mid P \mid Q \mid \\ & & P + Q \mid (\nu n)P \mid D(\bar{n}) \\ M, N & ::= & 0 \mid l\lfloor P \rfloor \mid M \parallel N \mid (\nu n)M \end{array}$$

In each one of the forms $a(n).P$, $(\nu n)P$ and $(\nu n)M$, the occurrences of n are binding. These lead to the notions of bound and free names as usual. We use $bn(P)/bn(M)$ and $fn(P)/fn(M)$ to denote the sets of bound and free names of P and M, respectively. Every identifier $D(\bar{n})$ has a definition $D(\bar{n}) \stackrel{def}{=} P$ where the $\bar{n}$ are pairwise distinct and $fn(P) = \{\bar{n}\}$[1]. As usual we require all free occurrences of $D(\bar{n})$ in P to be *guarded*; that is, they may only occur inside the continuation part Q of a guarded process $\pi.Q$ in P. We shall elide the parameters of a recursive process when they are unimportant or can be inferred from context. Substitutions are ranged over by σ, and we write $\{n/m\}$ for the substitution that maps m to n.

Structural congruence is given to identify processes up to trivial syntactic restructuring. It is defined for threads and agents, as $\equiv_l$ and $\equiv$, respectively as the smallest congruence satisfying the following rules:

- Structural congruence for threads, $\equiv_l$
  1. The Abelian monoid laws for Parallel $|$.
  2. The Abelian monoid laws for Sum $+$.
  3. $(\nu n)0 \equiv_l 0$, $(\nu n)(P+Q) \equiv_l P+(\nu n)Q$ if $n \notin fn(P)$, $(\nu n)(P|Q) \equiv_l P|(\nu n)Q$ if $n \notin fn(P)$
  4. $D(m) \equiv_l P\{m/n\}$ if $D(n) \stackrel{def}{=} P$
- Structural congruence for agents, $\equiv$
  1. $M \equiv N$ if $M \equiv_\alpha N$
  2. $m\lfloor P \rfloor \equiv m\lfloor Q \rfloor$ if $P \equiv_l Q$
  3. $M||0 \equiv M$, $M||N \equiv N||M$, $M||(N||L) \equiv (M||N)||L$
  4. The Abelian monoid laws for Parallel $\|$.

[1] Usually it is assumed that $fn(P) \subseteq \{\bar{n}\}$ for a process definition $D(\bar{n}) \stackrel{def}{=} P$, but we assume the stronger condition for convenience in technical discussion later.

ACT: $\pi \neq a(n) \Rightarrow \pi.P \xrightarrow{\pi} P$ **INP:** $a(n).P \xrightarrow{am} P\{m/n\}$

SUM: $P \xrightarrow{\alpha} P' \Rightarrow P+Q \xrightarrow{\alpha} P'$ **MAT:** $P \xrightarrow{\alpha} P' \Rightarrow [n=n]P \xrightarrow{\alpha} P'$

MISMAT: $P \xrightarrow{\alpha} P',\ n \neq m \Rightarrow [n \neq m]P \xrightarrow{\alpha} P'$ **TCOM:** $P \xrightarrow{\overline{a}\langle n\rangle} P',\ Q \xrightarrow{an} Q' \Rightarrow P \mid Q \xrightarrow{\tau} P' \mid Q'$

TPAR: $P \xrightarrow{\alpha} P',\ bn(\alpha) \cap fn(Q) = \emptyset,\ \alpha \neq \mathbf{split} \Rightarrow P \mid Q \xrightarrow{\alpha} P' \mid Q$

LACT: $P \xrightarrow{\alpha} P', \alpha \neq \tau, \mathbf{go}\ k, \mathbf{split} \Rightarrow l\lfloor P \rfloor \xrightarrow{\alpha,l} l\lfloor P' \rfloor$ **LTAU:** $P \xrightarrow{\tau} P' \Rightarrow l\lfloor P \rfloor \xrightarrow{\tau,l} l\lfloor P' \rfloor$

SPL: $P \xrightarrow{\mathbf{split},l} P' \Rightarrow l\lfloor P \mid Q \rfloor \xrightarrow{\tau,l} l\lfloor P' \rfloor \parallel l\lfloor Q \rfloor$ **GO:** $P \xrightarrow{\mathbf{go}\ k} P' \Rightarrow l\lfloor P \rfloor \xrightarrow{\tau,l} k\lfloor P' \rfloor$

MER: $l\lfloor P \rfloor \xrightarrow{\mathbf{merge},l} l\lfloor P' \rfloor \Rightarrow l\lfloor P \rfloor \parallel l\lfloor Q \rfloor \xrightarrow{\tau,l} l\lfloor P' \mid Q \rfloor$ **ACOM:** $M \xrightarrow{\overline{a}\langle n\rangle,l} M', N \xrightarrow{an,l} N' \Rightarrow M||N \xrightarrow{\tau,l} M'||N'$

APAR: $M \xrightarrow{\alpha,h} M',\ bn(\alpha) \cap fn(N) = \emptyset,\ \alpha \neq \mathbf{merge} \Rightarrow M \parallel N \xrightarrow{\alpha,h} M' \parallel N$

RES1: $M \xrightarrow{\alpha,l} M', n \notin n(\alpha), n \neq h \Rightarrow (\nu n)M \xrightarrow{\alpha,l} (\nu n)M'$ **RES2:** $M \xrightarrow{\alpha,l} M',\ l \notin n(\alpha) \Rightarrow (\nu l)M \xrightarrow{\alpha} (\nu l)M'$

OPEN: $M \xrightarrow{\overline{a}\langle n\rangle,l} M', a \neq n, n \neq l \Rightarrow (\nu n)M \xrightarrow{\overline{a}\langle \nu n\rangle,l} M'$ **STR:** $M \equiv N,\ M \xrightarrow{\alpha,h} M',\ M' \equiv N' \Rightarrow N \xrightarrow{\alpha,h} N'$

Table 1. Transition rules

Note that according to the structural congruence rules, the following terms are distinct: $l\lfloor P \rfloor \parallel l\lfloor Q \rfloor \not\equiv l\lfloor P \mid Q \rfloor$ and $l\lfloor 0 \rfloor \not\equiv 0$.

The operational semantics is given in terms of a labelled transition system, where each transition is labelled by *actions* and *locations*. There are seven kinds of *actions* α as follows: an output $\bar{a}\langle n\rangle$, an input an, a bound output $\bar{a}\langle \nu n\rangle$, the silent τ, a move **go** l, the join **merge** and the split action **split**. We write Act for the set of *actions*. Bound output action $\bar{a}\langle \nu n\rangle$ is a *bound action*; i.e. $bn(\bar{a}\langle \nu n\rangle) = \{n\}$ and $fn(\bar{a}\langle \nu n\rangle) = \{a\}$, and for all the other actions $bn(\alpha) = \emptyset$ and $fn(\alpha)$ is the set of the names occurring in α. The set of *locations* is defined as $Loc = \mathcal{N}_l \cup \{null\}$, ranged by $h, i, \cdots$. *null* is used in cases where concrete location names can not be specified.

A labelled transition system is a tuple: $(S, Act, Loc, \rightarrow)$, where S is a set of *states* and $\rightarrow \subseteq S \times Act \times Loc \times S$ is a *transition*. We write $M \xrightarrow{\alpha,h} M'$ to mean $(M, \alpha, h, M') \in \rightarrow$. We may omit the location label *null* in transitions; e.g., $M \xrightarrow{\tau} M'$ means $M \xrightarrow{\tau,null} M'$ The transition relations are defined by the rules in Table 1.

2.2 Bisimulation

Here we present a strong bisimulation called as "*LC-bisimulation (Location-conscious bisimulation)*", which equates processes whose both of action behaviours and locations where the actions occur are identical.

Definition A ground strong LC-bisimulation is a symmetric binary relation $\mathcal{S} \subseteq \mathcal{M} \times \mathcal{M}$, satisfying the following; $(M, N) \in \mathcal{S}$ then $\forall \alpha \in Act$ and $\forall h \in Loc$,

- $M \xrightarrow{\alpha,h} M' \wedge bn(\alpha) \notin fn(M, N)$ implies $\exists N' : N \xrightarrow{\alpha,h} N' \wedge (M', N') \in \mathcal{S}$

M and N are ground strong LC-bisimilar, written $M \dot{\sim}_l N$, if $(M, N) \in \mathcal{S}$ for some ground strong LC-bisimulation. Then strong LC-bisimilarity $\sim_l$ is defined by closing up $\dot{\sim}_l$ with substitutions; $M \sim_l N$ iff $M\sigma \dot{\sim}_l N\sigma$ for any σ. □

Proposition 1 *$\sim_l$ is an equivalence relation.* □

3 A predicate μ-calculus for Modular π-calculus

3.1 Syntax

Assume a countably infinite set $\mathcal{V}$ of *name variables*, ranged over by $x, y, \cdots$, which is disjoint from $\mathcal{N}$. We use $\eta, \mu, \ldots$ to range over $\mathcal{N} \cup \mathcal{V}$. We also assume a countable infinite set $\mathcal{X}$ of predicate variables, ranged over by $X, Y, \ldots$. Each predicate variable has associated with it an arity which is a natural number. The set of formulas of the logic is given by the following BNF grammar, where the formulas is divided into two categories, *propositions* (A) and *predicates*(F):

$$
\begin{array}{lcl}
\gamma & ::= & \bar{\eta}\langle\mu\rangle \mid \eta\mu \mid \tau \mid \mathbf{merge} \\
j & ::= & null \mid \eta \\
A & ::= & \neg A \mid A \wedge A \mid \langle\gamma@j\rangle A \mid \eta = \mu \mid 0 \mid \eta\lfloor 0 \rfloor \mid A|A \mid \\
 & & A||A \mid \eta \circledR A \mid \forall x.A \mid \mathrel{И}x.A \mid F(\bar{n}) \\
F & ::= & X \mid (\bar{x})A \mid \nu X.F
\end{array}
$$

The boolean connectives, name equality and universal quantifier ($\forall x.A$) have the standard meaning. $\mathrel{И}x.A$ is fresh-name quantifier; it is satisfied by an agent M if $A[n/x]$ is satisfied by M for every name which is fresh with respect to both M and A.

Action modalities are of the form $\langle\gamma@l\rangle$ where γ is an action and l a location. They are used to observe located transitions; i.e., an agent M satisfies $\langle\gamma@l\rangle A$ whenever $M \xrightarrow{\gamma,l} M'$ and M' has property A. Action γ is designed to be the actions observable at agent level;

$[\![\neg A]\!]_v = \mathcal{M} - [\![A]\!]_v$ $\quad$ $[\![\forall x.A]\!]_v = \bigcap_n [\![A[n/x]]\!]_v$

$[\![A \wedge B]\!]_v = [\![A]\!]_v \cap [\![B]\!]_v$ $\quad$ $[\![F(\bar{n})]\!]_v = [\![F]\!]_v(\bar{n})$

$[\![n = m]\!]_v = \mathit{if}\ n = m\ \mathit{then}\ \mathcal{M}$ $\quad$ $[\![X]\!]_v = v(X)$

$[\![\langle \gamma @ h \rangle A]\!]_v = \{M \mid \exists N \in [\![A]\!]_v . M \xrightarrow{\gamma,h} N\}$ $\quad$ $[\![(\bar{x})A]\!]_v = \lambda \bar{n}.[\![A[\bar{n}/\bar{x}]]\!]_v$

$[\![0]\!]_v = \{M \mid M \equiv 0\}$ $\quad$ $[\![\nu X.F]\!]_v = \sqcup\{f \mid f \sqsubseteq [\![F]\!]_{v[X \mapsto f]}\}$

$[\![n\lfloor 0 \rfloor]\!]_v = \{M \mid \exists M \in \mathcal{M}.M \equiv n\lfloor 0 \rfloor\}$ $\quad$ $[\![\text{Ⅵ}x.A]\!]_v = \bigcup_{n \notin (fn(A) \cup fn(M))} \{M \mid M \in [\![A[n/x]]\!]_v\}$

$[\![n®A]\!]_v = \{M \mid M \equiv (\nu n)N\ \mathit{and}\ N \in [\![A]\!]_v\}$ j

$[\![A \mid B]\!]_v = \{M \mid \exists n.M \equiv n\lfloor P_1 \mid P_2 \rfloor\ \mathit{and}\ n\lfloor P_1 \rfloor \in [\![A]\!]_v, n\lfloor P_2 \rfloor \in [\![B]\!]_v\}$

$[\![A \parallel B]\!]_v = \{M \mid M \equiv M_1 \parallel M_2\ \mathit{and}\ M_1 \in [\![A]\!]_v, M_2 \in [\![B]\!]_v\}$

Table 2. Denotation of name-closed formulas

that is, **go** l and **split** are excluded in γ since they are used internally within a system and not observable externally.

For the design of spatial constructs in the logic, common core constructs are chosen based on the design of *common core spatial logic* in [11]. But among them we exclude the *adjunct* operator for decidability of model checking algorithm discussed in Section 5. Thus, the spatial constructs in the logic are designed as *void* 0, *located void* $\eta\lfloor 0 \rfloor$, *thread composition* $A|B$, *agent composition* $A||B$ and *reveal* $\eta®A$. A characteristic of the design may be that spatial operators to observe void and compositions are prepared both for threads and agents. This design is for that the difference of processes between 0 and $l\lfloor 0 \rfloor$ and between $l\lfloor P|Q \rfloor$ and $l\lfloor P \rfloor || l\lfloor Q \rfloor$ are significant in $M\pi$ as they are not identified with structural congruence. On the contrary, reveal $n®A$ is prepared only for agents, since for example difference of processes such as $l[(\nu a)P]$ and $(\nu a)l[P]$ is not significant in $M\pi$, as identified with structural congruence. Hence it is sufficient to prepare such an operator to observe restrictions only for agents; i.e., restrictions for threads are observed by pulling them out to the agent level using rules in structural congruence.

Fixpoint formulas are expressed through predicates F. A predicate is either a predicate variable X, an abstraction $(\bar{x})A$ or a greatest fixpoint $\nu X.F$. The arity of a predicate F is defined thus: the arity of X if F has the form X or $\nu X.F$, or the length of $\bar{x}$ if F has the form $(\bar{x})A$. If $fnv(A) = \{\bar{x}\}$, the actual parameters to every free occurrences of X in A is $\bar{x}$, and X does not occur within the scope of any name quantifier, then we shall abbreviate $(\nu X.(\bar{x})A)(\bar{n})$ to $\nu X.A'[\bar{n}/\bar{x}]$ where A' is obtained from A by replacing $X(\bar{x})$ with X.

In both of quantification $\forall x.A$ and $\text{Ⅵ}x.A$, the occurrences of x are binding with scope A. Also abstraction $(\bar{x})A$ binds every variable in $\bar{x}$ with scope A. In $\nu X.F$ the predicate variable X is binding with scope F. Then these introduce the notions of *bound* and *free name variables* as well as *bound* and *free predicate variables* in the usual way. The set of free names, free name variables and free predicate variables of a formula A are respectively denoted by $fn(A)$, $fnv(A)$ and $fpv(A)$. We call formulas which do not have free name variables as *name-closed*, and formulas which have neither free name variables nor free names as *completely name-closed*. Formulas that do not have free predicate variables are *predicate-closed*. When forming an abstraction $(\bar{x})A$ it is required that $\bar{x}$ is a vector of distinct name variables, $fn(A) = \emptyset$ and $fnv(A) \subseteq \{\bar{x}\}$; i.e., abstractions are *completely name-closed*. As a consequence, all predicates are completely name-closed. Due to this, fixpoint formulas are *fully parameterized*, in the sense that the free names and free name variables of an application $F(\bar{n})$ are solely determined by the parameter part $\bar{n}$. We shall identify $((x)A)(u)$ with $A[u/x]$.

Negation $\neg$ is a negative operator. An occurrence of a predicate variable is *positive* if it is under an even number of negative operators. X occurs *positively* in a formula A/F if every occurrence of X in A/F is positive. Otherwise we say X occurs *negatively* in A/F.

A recursive formula $\nu X.F$ is *well-formed* if the arity of X is equal to that of F, F is completely name-closed and X occurs positively in F. A formula is *well-formed* if every recursive subformula of it is well-formed. We shall only consider well-formed formulas in this paper.

3.2 Satisfaction

The semantics of formulas is given to name-closed formulas, which may contain free predicate variables. To interpret them we need *predicate valuation*. A predicate valuation v assigns a function $v(X) : \mathcal{N}^k \to 2^{\mathcal{M}}$ to each predicate variable X of arity k. Modification of v at X by f is denoted by $v[X \mapsto f]$. For each k let $f \sqsubseteq^k g$ iff $f(\bar{n}) \subseteq g(\bar{n})$ for any $\bar{n} \in \mathcal{N}^k$. Then $(\{f : \mathcal{N}^k \to 2^{\mathcal{M}}\}, \sqsubseteq^k)$ is a complete lattice with meet $\sqcap^k$ and join $\sqcup^k$. We shall omit the superscript k and simply write $\sqcap$ and $\sqcup$, when no confusion may arise.

The denotation of formulas is defined in Table 2, where a proposition is interpreted as an element of $2^{\mathcal{M}}$, and a predicate with arity k as an element in the func-

tion space $\mathcal{N}^k \to 2^{\mathcal{M}}$. We shall write $M \models_v A$ to mean $M \in [\![A]\!]_v$.

An important property of the semantics is monotonicity:

Theorem 1 *(Monotonicity)*

1. *Suppose X is positive in A and F. Then $f \sqsubseteq g$ implies*
   (a) $[\![A]\!]_{v[X\mapsto f]} \subseteq [\![A]\!]_{v[X\mapsto g]}$
   (b) $[\![F]\!]_{v[X\mapsto f]} \sqsubseteq [\![F]\!]_{v[X\mapsto g]}$
2. *Suppose X is negative in A and F. Then $f \sqsubseteq g$ implies*
   (a) $[\![A]\!]_{v[X\mapsto g]} \subseteq [\![A]\!]_{v[X\mapsto f]}$
   (b) $[\![F]\!]_{v[X\mapsto g]} \sqsubseteq [\![F]\!]_{v[X\mapsto f]}$ □

Corollary 1 $[\![\nu X.F]\!]_v$ *is the greatest fixpoint of the function* $\lambda f.[\![F]\!]_{v[X\mapsto f]}$ □

Then we acquire the Gabbay-Pitts property:

Proposition 2 *(Gabbay-Pitts property) For name-closed formula* $И x.A$, $M \in [\![И x.A]\!]$ *iff* $M \in [\![A[n/x]]\!]$ *for every* $n \notin fn(M, A)$. □

3.3 Definable operators

3.3.1 Basic definable operators

The *true* operator is defined as $\mathbf{T} \triangleq 0 \vee \neg 0$, the *hidden name quantification* as $\mathrm{H}x.A \stackrel{def}{=} И x.x®A$, **1mod** as $\neg 0 \wedge \neg(\neg 0 \parallel \neg 0)$. $\langle - \rangle A$, which process M satisfies when any M' such that $M \xrightarrow{\gamma,h} M'$ satisfies A, is defined as: $\langle - \rangle A \triangleq \exists z.(\exists x \exists y.(\langle \bar{x}\langle y\rangle @ z\rangle A \vee \langle xy@z\rangle A) \vee \exists x.\mathrm{H}y.\langle \bar{x}\langle y\rangle @z\rangle A \vee \langle \mathbf{merge}@z\rangle A) \vee \langle \tau @ z\rangle A$. Similarly, $[-]A$, which process M satisfies when each M' such that $M \xrightarrow{\gamma,h} M'$ satisfies A, is defined as: $[-]A \triangleq \exists z.(\exists x \exists y.([\bar{x}\langle y\rangle @z]A \vee [xy@z\rangle A) \vee \exists x.\mathrm{H}y.[\bar{x}\langle y\rangle @z]A \vee [\mathbf{merge}@z]A) \vee [\tau @ z]A$. Then the *anytime* modality $\mathbf{AG}A$ can be defined with the greatest fixpoint: $\mathbf{AG}A \triangleq \nu X.A \wedge [-]X$. We extend negation to abstractions by letting $\neg(\bar{x})A = (\bar{x})\neg A$. Then the least fixpoint is defined by $\mu X.F \triangleq \neg \nu X.\neg F$. It is easy to check that the "sometime" modality can be defined as: $\mathbf{EF}A \triangleq \mu X.A \vee \langle - \rangle X$.

3.3.2 Spatial property for threads in an agent

In the proposed logic, spatial properties for threads inside an agent are not specified directly. For example, logical formula $l\lfloor \mathrm{H}x.(\langle \bar{x}\rangle \mathbf{T} \mid \langle x\rangle \mathbf{T})\rfloor$, which may be intuitively understood as "there is a secret connection between two threads within an agent at location l", is not a valid logical formula in the proposed logic in Section 3.1. However, a valid logical formula of the same meaning as the formula is expressed: $\mathrm{H}x.(\langle \bar{x}@l\rangle \mathbf{T} \mid \langle x@l\rangle \mathbf{T})$. Here we consider a simple logic to directly specify thread properties inside an agent together with an intuitive interpretation for them, and consider the denotation is given through translation function which translates the logic for threads to that in Section 3.1.

$$
\begin{array}{l}
t(l\lfloor \neg \dot{C}\rfloor) = \neg t(l\lfloor \dot{C}\rfloor) \wedge l\lfloor \mathbf{T}\rfloor \\
t(l\lfloor \dot{C} \wedge \dot{C}'\rfloor) = t(l\lfloor \dot{C}\rfloor) \wedge t(l\lfloor \dot{C}'\rfloor) \\
t(l\lfloor n = m\rfloor) = n = m \\
t(l\lfloor \langle \gamma\rangle \dot{C}\rfloor) = \langle \gamma @ l\rangle t(l\lfloor \dot{C}\rfloor) \\
t(l\lfloor \dot{0}\rfloor) = l\lfloor 0\rfloor \\
t(l\lfloor \dot{C} \mid \dot{C}'\rfloor) = t(l\lfloor \dot{C}\rfloor) \mid t(l\lfloor \dot{C}'\rfloor) \\
t(l\lfloor n®\dot{C}\rfloor) = n®t(l\lfloor \dot{C}\rfloor) \\
t(l\lfloor \forall x.\dot{C}\rfloor) = \forall x.t(l\lfloor \dot{C}\rfloor) \\
t(l\lfloor И x.\dot{C}\rfloor) = И x.t(l\lfloor \dot{C}\rfloor)
\end{array}
$$

Table 3. The translation function

The set of logical formulas to specify thread properties inside an agent is given by the following syntax:

$$
\begin{array}{lll}
C & ::= & l\lfloor \dot{C}\rfloor \\
\dot{C} & ::= & \neg \dot{C} \mid \dot{C} \wedge \dot{C} \mid \eta = \mu \mid \forall x.\dot{C} \mid \langle \gamma\rangle \dot{C} \mid 0 \mid \dot{C}|\dot{C} \mid \\
 & & n®\dot{C} \mid И x.\dot{C}
\end{array}
$$

The formula C is built with two-steps. First a subformula $\dot{C}$, which specifies thread properties, is defined. Each construct is intuitively interpreted in a similar manner as the logic in Section 3.1. Then the formula C is acquired by locating the subformula at some location. The formula above, i.e. $l\lfloor \mathrm{H}x.(\langle \bar{x}\rangle \mathbf{T} \mid \langle x\rangle \mathbf{T})\rfloor$, is an instance of such logic formulas.

Then a translation function t, which translates a logical formula for threads above to that in Section 3.1, is provided in Table 3. We can see the logical formulas for threads given with intuitive interpretation is expressed with the proposed logic in Section 3.1. In the table we use $l\lfloor \mathbf{T}\rfloor \triangleq l\lfloor 0\rfloor \vee l\lfloor \neg 0\rfloor$ and $l\lfloor \neg 0\rfloor \triangleq \langle - @ l\rangle A$.

Due to the function, for instance, the formula $l\lfloor \mathrm{H}x.(\langle \bar{x}\rangle \mathbf{T} \mid \langle x\rangle \mathbf{T})\rfloor$ is translated to the logic in section 3: $t(l\lfloor \mathrm{H}x.(\langle \bar{x}\rangle \mathbf{T} \mid \langle x\rangle \mathbf{T})\rfloor) = \mathrm{H}x.((\langle \bar{x}@l\rangle (l\lfloor 0\rfloor \vee l\lfloor \neg 0\rfloor) \mid \langle x@l\rangle (l\lfloor 0\rfloor \vee l\lfloor \neg 0\rfloor))$.

3.4 Examples

In Introduction, an example of specifying the cab system in $M\pi$ with the logic was given. In this subsection, we present a few more examples to illustrate the expressiveness of the logic.

First, the existence of a cab at some city, say "$city_a$", is specified using action modalities with the logical formulas as: $city_a\lfloor Cab\rfloor \triangleq \exists s.\langle call\ s@city_a\rangle \mathbf{T} \vee \langle \overline{run}@city_a\rangle \mathbf{T}$, specifying there is a process waiting

request via *call* or running ($\overline{run}$) at $city_a$. Then using spatial constructs, we may specify a property of "there are at least two cabs in $city_a$" as $city_a\lfloor Cab\rfloor || city_a\lfloor Cab\rfloor$. And by extending this, we may specify a property like "a traffic jam occurs in $city_a$", by observing many cabs, e.g., more than a hundred of cabs, running in the city, specified as: e.g., $\|_{100}$ $city_a\lfloor Cab\rfloor$, where we use $\|_k A$ to mean $\underbrace{A \| \cdots \| A}_{k}$.

Consider to extend the cab system to be constituted with many cabs and clients. For such a system, we may want consider a safety property such that "it is never possible for a cab anytime to carry more than two clients at one time". This property may be expressed by combining spatial operators and recursive formulas. Here the bad feature is that a busy cab is taking more than two clients, expressed as: $\exists city.(city\lfloor\langle\overline{run}\rangle\mathbf{T} \mid \neg 0 \mid \neg 0\rfloor)(\triangleq \exists city.(\langle\overline{run}@city\rangle\mathbf{T} \mid city\lfloor\neg 0\rfloor \mid city\lfloor\neg 0\rfloor))$, which simply says there are more than three threads within a modularity of a busy cab. Then the intended property may be specified as follows: $\mathbf{AG}(\neg\exists city.city\lfloor\langle\overline{run}\rangle\mathbf{T} \mid \neg 0 \mid \neg 0\rfloor)$.

4 Characterization

Consider two processes $l\lfloor\bar{a}\langle n\rangle|\bar{b}\langle n\rangle\rfloor$ and $l\lfloor\bar{a}\langle n\rangle.\bar{b}\langle n\rangle+\bar{b}\langle n\rangle.\bar{a}\langle n\rangle\rfloor$. These processes are equivalent with respect to a LC-bisimulation, but are distinguished by the formula $l\lfloor\neg 0|\neg 0\rfloor(=\langle -@l\rangle\mathbf{T}|\langle -@l\rangle\mathbf{T})$. Hence, we can find that these processes are, although LC-bisimilar, not logically equivalent. On the other hand, process $l\lfloor D(a,b)\rfloor$ with $D(x,y) \stackrel{def}{=} \bar{x}\langle y\rangle.D(x,y)$ and $l\lfloor D'(a,b)\rfloor$ with $D'(x,y) \stackrel{def}{=} \bar{x}\langle y\rangle.\bar{x}\langle y\rangle.D'(x,y)$ are LC-bisimilar, and in fact cannot be distinguished by any formula of the logic, where both processes denote the same single-threaded behaviour. However, they are not structurally congruent. In the following, we prove that the equivalence relation induced by the logic is coarser than structural congruence, and finer than strong LC-bisimulation.

Definition (Process logical equivalence) For processes M, N, we write $M =_L N$ if for all name-closed formulas A it holds that $M \models_v A$ *iff* $N \models_v A$. □

Proposition 3 *(1) If $M =_L N$, then $M \sim_l N$. (2) If $M \equiv N$, then $M =_L N$.* □

Here we put a remark on this result of characterization of logical equivalence, comparing it with those in [1, 3]. Although $M\pi$ is a variant of $D\pi$ (rather than a standard π-calculus), the result of characterization of the logical equivalence is closer to that in [1], where process model is π-calculus, than that in [3][2], where process model is a simple variant of $D\pi$. The result is partly due to the design of the process model and mainly due to logic design.

For the design of process model, $M\pi$ and [1] include the summation operator (+), while that in [3] does not. As the example above shows, the design of the process model with summation generates processes which are bisimilar but can be distinguished by the logics.

For the logic design, mainly that of the spatial constructs matters. The proposed logic is designed with action behavioural and spatial modalities. As stated in Subsection 2.1, the spatial constructs in the logic are designed based on "common core spatial constructs", and these operators observe the internal spatial structures of systems in a detailed manner. Such logic design makes distinguishing power of the logic stronger, and hence the equivalence the logic induces become more discriminating than bisimilarity. The logic design of [1] also includes such "common core spatial constructs". On the other hand, the logic in [3], as proposed as "a simple spatial logic", is designed so simply that only a few spatial constructs are equipped (and some common spatial constructs are absent). For instance, the logic does not include any spatial construct to observe restrictions (e.g. reveal $n\textcircled{R}A$), which is common in most of the other spatial logics. Such design of the logic as observes less spatial structures of systems makes the distinguishing power of the logic weaker, and hence the logical equivalence become closer to bisimilarity.

5 Model checking

First we consider a finite-state version of $M\pi$ (*finite-control* $M\pi$). It is designed based on *the finite-control* π [6], obtained by disallowing parallel composition through recursion. In $M\pi$ this restriction is not enough, since for the spatial characteristic of $M\pi$ the **split** operator in recursion may generate infinitely many reachable processes : e.g. $D \triangleq \mathbf{split}.D$ then $l\lfloor D\rfloor \xrightarrow{\tau,l} \cdots \xrightarrow{\tau,l} l\lfloor D\rfloor||l\lfloor 0\rfloor||\cdots||\lfloor 0\rfloor$. Hence, we also prohibit the **split** operator and parallel composition within recursion. Then a finite-control $M\pi$ is obtained as such.

Next we introduce a device to handle model-checking recursion processes with fix-point operators of logic, following the scheme of [19, 13, 14]. Given a finite set of names $\mathcal{N}_0 \subset \mathcal{N}$ and a finite set of closed processes $\mathcal{M}_0 \subseteq \mathcal{M}$, let $I = \{(\bar{n}_1, \bar{M}_1), \cdots, (\bar{n}_i, \bar{M}_i)\}$ where $\bar{n}_i$ are vector of the same length over $\mathcal{N}_0$, $n_i \neq n_j$ and $\bar{M}_i \subseteq \mathcal{M}_0$. I can be regarded as a function from

[2] In [3], it is concluded that the logical equivalence coincide with bisimilarity.

$check(M, \neg A) = \neg check(M, A)$ $\quad$ $check(M, A \wedge B) = check(M, A) \wedge check(M, B)$

$check(M, n = m) = \mathit{if}\ n = m\ \mathit{then\ true\ else\ false}$ $\quad$ $check(M, \langle \gamma @ h \rangle A) = \bigvee_{M \stackrel{\gamma:h}{\rightarrow} M'} check(M', A)$

$check(M, 0) = \mathit{if}\ M \equiv 0\ \mathit{then\ true\ else\ false}$ $\quad$ $check(M, n\lfloor 0 \rfloor) = \mathit{if}\ M \equiv n\lfloor 0 \rfloor\ \mathit{then\ true\ else\ false}$

$check(M, \text{Ⓝ}x.A) = check(M, A[newname()/x])$

$check(M, A|B) = \bigvee_{(M_1, M_2) \in decomp^{|}(M)} check(M_1, A) \wedge check(M_2, B)$

$check(M, A||B) = \bigvee_{(M_1, M_2) \in decomp^{\|}(M)} check(M_1, A) \wedge check(M_2, B)$

$check(M, n\text{Ⓡ}A) = \mathit{if}\ n \in fn(M)\ \mathit{then\ false,\ else} \bigvee_{M \prec_n M'} check(M', A) \vee check(M, A)$

$check(M, \forall x.A) = (\bigwedge_{n \in fn(A)} check(M, A[n/x])) \wedge check(M, A[newname()/x])$

$check(M, (\nu X.\{I\}F)(\tilde{n})) = \mathit{if}\ M \in I(\tilde{n})\ \mathit{then\ true\ else}\ check(M, F[\nu X.\{\{(\tilde{n}, M)\} \sqcup I\}F/X](\tilde{n}))$

Table 4. The model checking algorithm

$\mathcal{N}^k$ to $2^{\mathcal{M}}$, where k (the arity of I) is the length of $\tilde{n}_1$, by letting $I(\tilde{n}) = \{\bar{M}\}$ if for some $\bar{M}$ s.t. $(\tilde{n}, \bar{M}) \in I$ and $I(\tilde{n}) = \emptyset$ otherwise. Then we generalize the syntax of fixpoint predicates to $\nu X\{I\}F$, where the arity of I equals to that of X, and regard $\nu X.F$ as an abbreviation of $\nu X\{\emptyset\}.F$. The $\{I\}$ component is a history-recording device to facilitate loop detection during model checking. The interpretation of the generalized fixpoint predicates is $[\![\nu X.\{I\}F]\!]_v = \sqcup\{f \mid f \sqsubseteq [\![F]\!]_v[X \mapsto f]\}$. Lemma 1 provides a termination strategy to decide when to stop the model checking a recursively defined property.

Lemma 1 *If $M \in I(\tilde{n})$ then $M \in [\![\nu X.\{I\}F]\!]_v(n)$; Otherwise $M \in [\![\nu X.\{I\}F]\!]_v(\tilde{n})$ iff $M \in [\![F[\nu X.\{(\tilde{n}.\{M\})\}F/X]\!]_v(\tilde{n})$.* □

An agent is *normal* if all bound names are distinct and all *unguarded* restrictions are at the agent level and at the top level. A normal form $(\nu n_1, ..., n_k)M$ is a strict normal form if $n_i \in fn(M)$ for any $1 \leq i \leq k$. We denote the normal form and the strict normal form of M by nf(M) and snf(M), respectively. If $snf(M) = (\nu n_1, ..., n_i)N$ then we write $M \prec_n N$ for any $n \in \{n_1, ..., n_i\}$. If $\tilde{n} = \{n_1, \cdots, n_i\}$ then we shall abbreviate $(\nu n_1, \cdots, n_i)M$ to $(\nu\tilde{n})M$. If $J = \{1, \cdots, i\}$ then we shall abbreviate agents $M_1 \parallel \cdots \parallel M_i$ to $\|_{j \in J}\ M_j$, and threads $P_1 \mid \cdots \mid P_i$ to $|_{j \in J}\ P_j$. The symbol ⊎ stands for disjoint union. Then $decomp^{\|}(M)$ denotes the set $\{(\nu\tilde{n}_1)(\|_{j \in J_1}\ M_1), (\nu\tilde{n}_2)(\|_{j \in J_2}\ M_2) : snf(M) = (\nu\tilde{n})(\|_{j \in J}\ M_j), \tilde{n} = \tilde{n}_1 \uplus \tilde{n}_2, J = J_1 \uplus J_2\}$. Function $decomp^{|}(M)$, where M is an agent of a single process modularity of the form $(\nu\tilde{n})l\lfloor P \rfloor$, denotes the set $\{(\nu\tilde{n}_1)(l\lfloor|_{j \in J_1}\ P_1\rfloor), (\nu\tilde{n}_2)(l\lfloor|_{j \in J_2}\ P_2)\rfloor) : snf(M) = (\nu\tilde{n})(l\lfloor|_{j \in J}\ P_j\rfloor), \tilde{n} = \tilde{n}_1 \uplus \tilde{n}_2, J = J_1 \uplus J_2\}$.

The model checking algorithm is presented in Table 4. In the algorithm, function *newname* returns the least name currently not used in the set of name $N_{M,A} = \{n_1, \cdots, n_{\sharp_M + \sharp_A + 1}\}$, where we let $\sharp_M$ be the number of names appearing in M and $\sharp_A$ the number of names and name variables appearing in A.

Note that in the case of $check(M, \langle \gamma @ h \rangle A)$, although for each $\langle \gamma @ h \rangle$-derivation from M there may be infinite many derivatives M', all of these derivatives are structurally congruent, thus it suffices to use one of them as a representative due to Proposition 3-(2). Therefore, disjunction is finite.

Theorem 2 *For any finite-control process M and logical formula A, **check**(M, A) always terminate and it returns true if and only if $M \models A$.* □

To show the usefulness of the model checking algorithm, we demonstrate an example, where the algorithm actually checks if the system of cab protocol satisfies the property given in Introduction. That is, we calculate the following:

$$check(CP, (\langle \overline{call} \langle city_d \rangle @ city_a \rangle \mathbf{T} \parallel \exists s.\langle call\ s @ city_a \rangle \mathbf{T}) \Rightarrow \mathbf{EF}\langle \overline{walk} @ city_d \rangle \mathbf{T})$$

Now we have

$$(\langle \overline{call} \langle city_d \rangle @ city_a \rangle \mathbf{T} \parallel \exists s.\langle call\ s @ city_a \rangle \mathbf{T}) \Rightarrow \mathbf{EF}\langle \overline{walk} @ city_d \rangle \mathbf{T}$$
$$= \neg(\langle \overline{call} \langle city_d \rangle @ city_a \rangle \mathbf{T} \parallel \exists s.\langle call\ s @ city_a \rangle \mathbf{T}) \vee (\mu X.(x, y)(\langle \overline{x} @ y \rangle \mathbf{T} \vee \langle - \rangle X(x, y))(walk, city_d))$$

and

$$\mu X.(x, y)(\langle \overline{x} @ y \rangle \mathbf{T} \vee \langle - \rangle X(x, y)) = \neg\nu X.(x, y)\neg(\langle \overline{x} @ y \rangle \mathbf{T} \vee \langle - \rangle \neg X(x, y)).$$

Then, it suffices to calculate the following:

$$check(CP, \neg(\langle \overline{call} \langle city_d \rangle @ city_a \rangle \mathbf{T} \parallel \exists s.\langle call\ s @ city_a \rangle \mathbf{T}) \vee \neg(\nu X.(x, y)\neg(\langle \overline{x} @ y \rangle \mathbf{T} \vee \langle - \rangle X(x, y))(walk, city_d)).$$

A part of the demonstration where the model checking algorithm automatically proves that the cab system CP actually satisfies the property presented in Introduction is shown in Table 5, where we use the following abbreviation: $F \stackrel{abv}{=} (x, y)\neg(\langle \overline{x} @ y \rangle \mathbf{T} \vee \langle - \rangle \neg X(x, y))$.

$$
\begin{aligned}
&check(CP, \neg(\langle\overline{call}\langle city_d\rangle@city_a\rangle\mathbf{T} \parallel \exists s.\langle call\ s@city_a\rangle\mathbf{T}) \\
&\qquad \vee\neg\nu X.(x,y)\neg(\langle\overline{x}@y\rangle\mathbf{T} \vee \langle-\rangle\neg X(x,y)(walk, city_d)) \\
&= check(CP, \neg(\langle\overline{call}\langle city_d\rangle@city_a\rangle\mathbf{T} \parallel \exists s.\langle call\ s@city_a\rangle\mathbf{T})) \\
&\qquad\qquad \vee check(CP, \neg(\nu X.F)(walk, city_d)) \\
&= false \vee check(CP, \neg(\nu X.F)(walk, city_d)) \\
&= check(CP, \neg F[\nu X\{CP\}.F/X](walk, city_d)) \\
&= check(CP, \\
&\neg(\neg(\langle\overline{walk}@city_d\rangle\mathbf{T} \vee \langle-\rangle\neg(\nu X\{CP\}.F)(walk, city_d))) \\
&= false \vee check(CP, \langle-\rangle\neg(\nu X\{CP\}.F)(walk, city_d)) \\
&= check(E_1, \neg(\nu X\{CP\}.F)(walk, city_d)) \\
&= check(E_1, \neg F[\nu X\{CP, E_1\}.F/X](walk, city_d)) \\
&= check(E_1, \neg(\neg(\langle\overline{walk}@city_d\rangle\mathbf{T} \vee \\
&\qquad\qquad \langle-\rangle\neg\nu(X\{CP, E_1\}.F)(walk, city_d))) \\
&= check(E_1, \langle-\rangle\neg(\nu X\{CP, E_1\}.F)(walk, city_d))) \\
&\cdots \\
&= check(E_2, \neg F[\nu X\{CP, E_1, E_2\}.F/X](walk, city_d)) \\
&\cdots \\
&= check(E_5, (\langle\overline{walk}@city_d\rangle\mathbf{T} \vee \\
&\qquad \langle-\rangle\neg(\nu X\{CP, E_1, E_2, E_3, E_4\}.F)(walk, city_d))) \\
&= check(E_5, \langle\overline{walk}@city_d\rangle\mathbf{T}) \vee check(E_5, \cdots \\
&= true
\end{aligned}
$$

Table 5. Model checking example

6 Conclusion and future work

We have presented a modal logic for Mπ: a logic to specify properties for systems in Mπ. Characterization of process equivalence the logic induce has been investigated, and shown that the distinguishing power of the logic falls between bisimilarity and structural congruence. A model checking algorithm for the logic over the finite-control subset of $M\pi$ has been proposed, and its correctness proved. We also have demonstrated several examples to show the usefulness of the logic and the model checking algorithm.

For future work, we would like to consider a minimal design of the logic following [11]. Another avenue for future research is to improve the efficiency of the model checking algorithm.

References

[1] L. Caires. Behavioral and spatial observations in a logic for the π-calculus. In *Proceeding of FoSSaCS 2005*, LNCS 2987. Springer-Verlag, 2005.

[2] L. Caires and L. Cardelli. A spatial logic for concurrency. In *Proceeding of 4th International Symposium of Theoretical Aspects of Computer Software(TACS)*, volume 2215 of *LNCS*, pages 1–37. Springer, 2001.

[3] L. Caires and H. T. Vieira. Extensionality of spatial observations in distributed systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 175:131–149, 2007.

[4] L. Cardelli and A. D. Gordon. Ambient logic. *Mathematical Structures in Computer Science*, 2006.

[5] W. Charatonik, A. Gordon, and J. M. Talbot. Finite-control mobile ambients. In *Proceedings of the 11th European Symposium on Programming Languages and Systems(ESOP)*, volume 2305 of *LNCS*, pages 295–313, 2002.

[6] M. Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 1996.

[7] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM (JACM)*, 32, 1985.

[8] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Information and Computation*, volume 173, pages 82–120, 2002.

[9] D. Hirschkoff. An extensional spatial logic for mobile processes. In *Proceeding of CONCUR 2004*, volume 3170 of *LNCS*. Springer-Verlag, 2004.

[10] D. Hirschkoff, E. Lozes, and D. Sangiorgi. Separability, expressiveness, and decidability in the ambient logic. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 423–432. IEEE Computer Society,, 2002.

[11] D. Hirschkoff, E. Lozes, and D. Sangiorgi. Minimality results for the spatial logics. In *Foundations of Software Technologyand Theoretical Computer Science*, LNCS 2914. Springer-Verlag, 2003.

[12] T. Kitamura and H. Lin. Controlling process modularity in mobile computing. In *Proceeding of ICTAC'07*, volume 1686 of *LNCS*, pages 246–259. Springer, 2007.

[13] H. Lin. A predicate spatial logic and model checking for mobile processes. In *Proceeding of ICTAC'04*, volume 3407 of *LNCS*, 2004.

[14] H. Lin. A predicate μ-calculus for mobile ambients. *Journal of Computer Science and Technology*, 20, 2005.

[15] R. Milner, J. Parrow, and D.Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114, 1993.

[16] D. Sangiorgi. Extensionality and intensionality of the ambient logics. In *Proceeding of 28th Annual Symposium on Principles of Programming Languages*. ACM, 2001.

[17] C. Stirling. *Modal and Temporal Properties of Processes (Texts in Computer Science)*. Springer, 2001.

[18] E. Tuosto and H. T. Vieira. An observational model for spatial logics. *Electronic Notes in Theoretical Computer Science*, 142:229–254, 2006.

[19] G. Winskel. A note on model checking the modal μ-calculus. *Theoretical Computer Science*, 83, 1991.