Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 123 (2005) 19–33
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Modal Logic for π-Calculus and Model Checking Algorithm [1]

Taolue Chen [2]

State Key Laboratory of Novel Software Technology
Nanjing University, Nanjing, P.R.China

Tingting Han

State Key Laboratory of Novel Software Technology
Nanjing University, Nanjing, P.R.China

Jian Lu

State Key Laboratory of Novel Software Technology
Nanjing University, Nanjing, P.R.China

Abstract

The π-calculus is one of the most important mobile process calculi and has been well studied in the literatures. Temporal logic is thought as a good compromise between description convenience and abstraction and can support useful computational applications, such as model-checking. In this paper, we use symbolic transition graph inherited from π-calculus to model concurrent systems. A wide class of processes, that is, the finite-control processes can be represented as finite symbolic transition graph. A new version π-μ-Logic is introduced as an appropriate temporal logic for the π-calculus. Since we make a distinction between proposition and predicate, the possible interactions between recursion and first-order quantification can be solved. A concise semantics interpretation for our modal logic is given. Based on the above work, we provide a model checking algorithm for the logic, which follows the well-known Winskel's tag set method to deal with fixpoint operator. As for the problem of name instantiating, our algorithm follows the 'on-the-fly' style, and systematically employs schematic names. The correctness of the algorithm is shown.

Keywords: π-calculus, Symbolic Transition Graph, π-μ-logic, Model Checking Algorithm.

[1] Supported by 973 Program of China (2002CB312002), NNSFC (60273034, 60233010), JSFC (BK2002203, BK2002409)
[2] Email: ctl@ics.nju.edu.cn

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2004.04.043

1 Introduction

Over the last decades, various calculi of mobile processes, notably the π-calculus [13], have been the focus of research in concurrency theory. Because of the deficiency of using algebra method to model and describe related properties of systems, e.g. mobility, safety, a lot of research has focused on modal logic of calculus. Modal logic (temporal logic especially) is thought as a good compromise between description convenience and abstraction. In addition, many modal logics support useful computational applications, such as model-checking. As a powerful language to describe mobile and dynamic process networks, the problem of verifying general temporal and functional properties, cast in terms of the π-calculus, has been investigated in-depth. Some modal logic systems for π-calculus have been provided in the literatures. The original work, as far as we know, belongs to Milner et al. In [14], they provided a cluster of extensions for Hennessy-Milner Logic [8], and proved that two of them characterize the two bisimulation equivalences, that is, the strong late and early bisimulation. However, their extension is rather simple. This may be owned to their motivation to characterize bisimulaiton relation. In [1][4] and more recently [6], Amadio and Dam introduced recursion into the modal logic via fixpoints, as in the propositional μ-calculus, thus has the ability to express properties for processes with infinite behaviors. These logic systems may be referred as π-μ-calculus. The main concern of these two papers is to formulate proof systems for deriving statements asserting whether a process satisfies a formula. What's more, from our point of view, although the composition proof systems in the two papers are subtle, they are a little tedious, especially the completeness proof. We think it is rooted in the lacking of adequate 'symbolic' information. The start point is to remedy this deficiency in some sense. It is well-known that symbolic technique has been widely used for name-passing calculi, especially providing the complete proof system for bisimulation equivalence and devising efficient bisimulation checking algorithm, e.g. [10][9]. In this paper, we borrow the ideas from this technique and adapt it to devising model checking algorithms.

We present our main idea in brief. In this paper, first, we use symbolic transition graph to model concurrent systems. A wide class of processes, that is, the finite-control processes can be represented as a finite symbolic transition graph. And the transition from process terms to symbolic transition graph is direct and rather simple. Second, we introduce a new version π-μ-Logic, an extension of the modal μ-calculus with boolean expressions over names, and primitives for name input and output as an appropriate temporal logic for the π-calculus. Note that in our π-μ-logic, the 'bound output' modality is worth paying attention to. The fresh name quantification due to Pitts which is

used in spatial logic [2] is subsumed implicitly, thus we must face the problem of possible interactions between recursion and first-order quantification. To solve this problem, we make a distinction between proposition and predicate in the syntax of logic system, thus a concise semantics interpretation for our modal logic can be given while the notion of 'property sets' is not needed. We defer more details to Section 4. The main contribution of our work lies in the model checking algorithm for the logic introduced in this paper. We follow the well-known Winskel's tag set method to deal with fixpoint operator since we prefer the local algorithm. As for the problem of name instantiating, our algorithm follows the 'on-the-fly' style, and systematically employs the so called schematic names, that is, the fresh name set of current node and logical formula with one new name. The correctness of the algorithm is shown.

The rest of the paper is organized as follows: some background material for π-calculus, especially the symbolic transition graph is reviewed in the following section. In Section 3, the π-μ-logic is introduced and the semantics is given, some useful properties are also discussed in this section. The model checking algorithm is presented and its correctness is shown in Section 4. The paper is concluded with Section 5 where related work is also discussed. Note that in this extended abstract, due to space restriction, most of the detailed proofs are omitted. We refer the interested readers to the full version of the paper.

2 π-calculus and Symbolic Transition Graph

In this section, we review some background knowledge on π-calculus and introduce the notion of symbolic transition graph.

Boolean Expressions and Substitution

The basic entities of the π-calculus are names, i.e. identifiers for communication channels. Let $\mathcal{N}$, ranged over by $a, b, m, n \ldots$ be a countably infinite set of names. Vectors of names will be denoted by $\tilde{a}, \tilde{b}, \tilde{m}, \tilde{n} \ldots$.

Boolean Expressions, ranged over by ϕ, ψ, are defined by BNF as follows:

$$\phi ::= true \mid x = y \mid \neg\phi \mid \phi \wedge \phi$$

We will write $BExp$ for the set of boolean expressions and we use $false$, $x \neq y$ and $\phi \vee \psi$ to denote $\neg true$, $\neg(x = y)$ and $\neg(\neg\phi \wedge \neg\psi)$.

The evaluation of a boolean formula Ev is a function $Ev : BExp \rightarrow$

$\{true, false\}$, and is defined as follows:

$$Ev(x = x) = true \qquad Ev(x = y) = false \text{ if } x \not\equiv y$$
$$Ev(\neg\phi) = \neg Ev(\phi) \qquad Ev(\phi \wedge \psi) = Ev(\phi) \wedge Ev(\psi)$$

Substitutions, ranged over by σ, δ, etc, are partial mappings from $\mathcal{N}$ to $\mathcal{N}$. If $\sigma = [\tilde{y}/\tilde{x}]$, where the length of $\tilde{x}$ equals to that of $\tilde{y}$, then $dom(\sigma) = \{\tilde{x}\}$, $cod(\sigma) = \{\tilde{y}\}$ and $n(\sigma) = \{\tilde{x}\} \cup \{\tilde{y}\}$. If $fn(\phi) \subseteq V$, we say ϕ is a boolean expression on V. Note that σ maps x onto y for $x \in dom(\sigma)$ and x onto itself for $x \notin dom(\sigma)$. In the sequel, we will use $\emptyset$ to denote empty substitution, $\sigma\delta$ to denote the composition of σ and δ. The substitution $\sigma[x \mapsto z]$ is the same as σ except that it maps x to z instead of $x\sigma$. The restriction of σ on $V \subseteq \mathcal{N}$, written $\sigma_{|V}$ is defined as if $x \in V$ then return $x\sigma$ else return x itself.

For each name x, the function ν_x is defined in a stardand way in literature. We refer the reader to [10] for details.

A substitution σ satisfies ϕ, written $\sigma \models \phi$, if $Ev(\phi\sigma) = true$. We write $\phi \Rightarrow \psi$ to mean that $\sigma \models \phi$ implies $\sigma \models \psi$ for any substitution σ, and $\phi = \psi$ to mean $\phi \Rightarrow \psi$ and $\psi \Rightarrow \phi$. ϕ is consistent if there are no $x, y \in \mathcal{N}$ s.t. $\phi \Rightarrow [x = y]$ and $\phi \Rightarrow [x \neq y]$ at the same time. Otherwise it is inconsistent. It is easy to see that ϕ is consistent iff there exists a substitution σ, s.t. $\sigma \models \phi$. ϕ is valid, if $Ev(\phi) = true$. Note for a valid boolean expression ϕ, we have $\sigma \models \phi$ for any substitution σ, thus it can be denoted by $true$.

Substitutions that just interchange a pair of names, which is called transpositions and ranged by θ, will play a special role in the following technical developments. More precisely, the transposition of n and m, written as $\{m \leftrightarrow n\}$, denotes the substitution $\sigma : \{m, n\} \rightarrow \{n, m\}$. It turns out that transposition is a useful tool in proving properties concerning fresh names.

Symbolic Transition Graph

Since π-calculus and related notions is well-known, we will omit the detailed presentation on its syntax in order to save space. Readers who are not familiar with it can refer to some standard literature, e.g. [13]. We only point out that in this paper, we disallow the parallel composition operator $|$ to appear in the bodies of recursive definitions, and call this restricted language 'finite-control' π-calculus [5]. By confining the process expression to *finite control* processes, which is the syntactic counterpart of CCS finite state processes, the symbolic transition graph in the below is finite thus we obtain a decidable model checking problem.

Now, we introduce the notion of *Symbolic Transition Graph* (STG for short) as a new model for π-calculus process terms. In the sequel, let $SAct =$

$\{\tau\} \cup \{a(b), \bar{a}b, \bar{a}(b) \mid a, b \in \mathcal{N}\}$ denote the set of *symbolic actions*. For a set of names $V \subset_{fin} \mathcal{N}$, function new$(V)$ returns the least name in $\mathcal{N} \backslash V$.

Definition 2.1 (**Symbolic Transition Graph**) A symbolic transition graph (STG for short) is a rooted directed graph where each node n has an associated finite set of free names $fn(n)$ and each edge is labelled by a tuple (ϕ, α) where ϕ is the boolean expression $\phi \in BExp$ and $\alpha \in SAct$ is a symbolic action. A STG is *well-formed* if where (ϕ, α) is the label of an edge from n to m, written $n \overset{\phi,\alpha}{\mapsto} m$, then $fn(\phi) \subseteq fn(m)$ and $fn(n) \subseteq fn(m) \cup bn(\alpha)$.

We write $m \overset{true,\alpha}{\mapsto} n$ as $m \overset{\alpha}{\mapsto} n$ for simplicity. Given a process t in π-calculus, the corresponding STG can be generated by some systematic rules. Due to space restriction, we omit the details.

It is worth pointing out that for finite-control process t, by which the symbolic transition graph is generated is finite. Instead of giving the operation semantics for π-calculus terms, we give the concrete operational semantics for the STGs. First, we introduce some notations. Given a STG, a state n_σ is a pair consisting of a node n together with a substitution σ associated with it. The set of free names $fn(n_\sigma)$ is defined as $fn(n)\sigma$ and it is understood that σ is restricted to $fn(n)$.

The late (concrete) operational semantics is defined as the least relation over states generated by the rules as follows:

$$\frac{m \overset{\phi,\tau}{\mapsto} n}{m_\sigma \overset{\tau}{\rightarrow} n_\sigma} \quad \sigma \models \phi \qquad \frac{m \overset{\phi,\bar{a}b}{\mapsto} n}{m_\sigma \overset{\bar{a\sigma}b\sigma}{\rightarrow} n_\sigma} \quad \sigma \models \phi$$

$$\frac{m \overset{\phi,a(b)}{\mapsto} n}{m_\sigma \overset{a\sigma(c)}{\rightarrow} n_{\sigma[b \mapsto c]}} \quad \sigma \models \phi \wedge c \notin fn(m_\sigma) \qquad \frac{m \overset{\phi,\bar{a}(b)}{\mapsto} n}{m_\sigma \overset{\bar{a\sigma}(c)}{\rightarrow} n_{\sigma[b \mapsto c]}} \quad \sigma \models \phi \wedge c \notin fn(m_\sigma)$$

An important property of label transition is the so called monotonicity property. It is well known when mismatch is included in the calculus, the common presentation of this property (using substitution) does not hold. However, when transposition is used, we have:

Lemma 2.2 *Given STG $\mathcal{G}$, s, s' are nodes of $\mathcal{G}$, θ is a transposition, the following properties hold:*

(i) If $s \overset{\alpha}{\rightarrow} s'$, then $s\theta \overset{\alpha\theta}{\rightarrow} s'\theta$.

(ii) If $s\theta \overset{\alpha}{\rightarrow} s'$, then there exists α', s'', such that $s \overset{\alpha'}{\rightarrow} s''$, $\alpha'\theta \equiv \alpha$ and $s''\theta \equiv s$.

Proof. By structure induction on transition. □

3 π-μ-Logic

Syntax

We assume a countably infinite set $\mathcal{V}$ of name variables ranged over by $x, y, z \ldots$, such that $\mathcal{V} \cap \mathcal{N} = \emptyset$. And we assume a countably infinite set $\mathcal{X}$ of *predicate variables*, ranged over by $X, Y, Z, \ldots$. Each predicate variable has been assigned an *arity* $n \in \omega$, written $X : n$. The syntax of the formula is defined by BNF as follows:

$$\begin{aligned} \alpha &::= \tau \mid u?(x) \mid u?v \mid u!v \mid u!(x) \\ \phi &::= true \mid u = v \\ A, B &::= \phi \mid \Lambda(\tilde{u}) \mid \neg A \mid A \wedge B \mid \forall x.A \mid \langle \alpha \rangle A \\ \Lambda &::= X \mid (\tilde{x})A \mid \nu X.\Lambda \end{aligned}$$

where, $u, v \in \mathcal{N} \cup \mathcal{V}$. The syntax is divided into two categories: *propositions* and *predicates*. Semantically, propositions denote sets of nodes in a STG (i.e. process terms), while predicates denote functions from sets of names to sets of nodes. For propositions, the operators are rather standard since it is adapted from well-known Hennessy-Milner Logic [8]. A predicate is either a predicate variable X, or an abstraction $(\tilde{x})A$, or a greatest fixpoint $\nu X.\Lambda$. When forming an abstraction $(\tilde{x})A$, as our notation indicates, it is required that $\tilde{x}$ be a vector of distinct **name variables**. Then the arity of a predicate Λ is defined as: the arity of X if Λ has the form X or $\nu X.\Lambda$, or the length of $\tilde{x}$ if Λ has the form $(\tilde{x})A$. In abstractions and applications we always require arities to be matched properly. In formulas of the form $\forall x.A$, $\langle u?(x) \rangle.A$, $\langle u!(x) \rangle.A$, $(\tilde{x})A$ and $\nu X.\Lambda$, the distinguished occurrences of x and X are binding, with the scope of the propositions A or predicate Λ. 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 denoted by $fn(A)$, $fnv(A)$ and $fpv(A)$ respectively. Formulas that do not have free name variables are **name-closed**. Formulas that do not have free predicate variables are **predicate-closed**. A formula is **closed** if it is both name-closed and predicate-closed.

We defined on formulas the relation $\equiv_\alpha$ of α-congruence in the standard way, that is, as the least congruence identifying formulas modulo renaming of bound (name and predicate) variables. We will consider formulas always modulo α-congruence. Note that for formula, the notion of name substitution is extended to function from $\mathcal{N} \cup \mathcal{V}$ to $\mathcal{N}$, i.e. we allow the name variables to be replaced by names. Note that for convenience, we identify β-equivalence formulas, that is, $((\tilde{x})A)(\tilde{u})$ and $A[\tilde{u}/\tilde{x}]$.

The unary operator $\neg$ is negative. 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 if every occurrence of X in A is positive. Otherwise we say X occurs negatively in A. A fixpoint predicate $\nu X.\Lambda$ is **well-formed** if $fn(\Lambda) = fnv(\Lambda) = \emptyset$ and X occurs positively in Λ. Note that we require that predicate Λ has no **free name**, thus $n(\Lambda(\tilde{u}))$ and $fv(\Lambda(\tilde{u}))$ are totally determined by the actual parameter $\tilde{u}$, which is very important to the soundness of semantics. A formula is well-formed if every fixpoint subformula in it is well-formed. In the sequel, we only consider well-formed formulas. Note that as usual, in our π-μ-logic system, we can define some standard derived connectives. In this paper, we choose an economical way to present our logical system.

Semantics

Given STG G, by the concrete operational semantics rules, we can get a concrete graph denoted by $\mathcal{G}$. The semantics of formula is defined by assigning to each formula A a node set of $\mathcal{G}$, i.e. $[\![A]\!]$, namely all the nodes of $\mathcal{G}$ that satisfy the property denoted by A. For convenience, we denote n_σ by s and for any $s \equiv n_\sigma$, $s[c/b] \equiv n_{\sigma[b\mapsto c]}$. Since formulas may contain free name variables and free predicate variables, to interpret them we need name valuations and predicate valuations. A name valuation ρ is an extended version of substitution, which is a total mapping from $\mathcal{N} \cup \mathcal{V} \to \mathcal{N}$ with identity on $\mathcal{N}$. A predicate valuation ξ assigns to every predicate variable X of arity k a function $\xi(X) : \mathcal{N}^k \to \wp(\mathcal{G})$. As usual, the relation $\subseteq$ can be extended point-wise to functional space as follows: for each k, two functions $f^{(k)}, g^{(k)} : \mathcal{N}^k \to \wp(\mathcal{G})$, define $f^{(k)} \sqsubseteq g^{(k)}$ *iff* $f(\tilde{n}) \subseteq g(\tilde{n})$ for any $\tilde{n} \in \mathcal{N}^k$. Thus, the functional space $\mathcal{N}^k \to \wp(\mathcal{G})$ forms a complete lattice w.r.t. $\sqsubseteq$. The denotation of formulas is defined inductively in Fig.1.

If A is name-closed then $[\![A]\!]_{\rho;\xi}$ does not depend on ρ and will be written $[\![A]\!]_\xi$. Furthermore, if A is name-closed and predicate-closed, then $[\![A]\!]_{\rho;\xi}$ depends on neither ρ nor ξ and it will be written as $[\![A]\!]$. We will write $s \models A$ to denote $s \in [\![A]\!]$.

As in the case of first-order logic, the following lemma which relates substitutions with valuations is common, and will be used implicitly.

Lemma 3.1 *The following properties hold:*

(i) $[\![A[b/x]]\!]_{\rho;\xi} = [\![A]\!]_{\rho[x\mapsto b];\xi}$.

(ii) $[\![\Lambda[F/X]]\!]_{\rho;\xi} = [\![\Lambda]\!]_{\rho;\xi[X\mapsto\xi(F)]}$.

Proof. By mutual induction on the structure of A and Λ. □

It is routine to show that for any formula A and B with $A \equiv_\alpha B$, $[\![A]\!]_{\rho,\xi} =$

$$
\begin{aligned}
[\![\phi]\!]_{\rho;\xi} &= \begin{cases} \mathcal{G} & \text{If } \rho \models \phi \\ \emptyset & \text{o.w.} \end{cases} \\
[\![A \wedge B]\!]_{\rho;\xi} &= [\![A]\!]_{\rho;\xi} \cap [\![B]\!]_{\rho;\xi} \\
[\![\neg A]\!]_{\rho;\xi} &= \mathcal{G} \backslash [\![A]\!]_{\rho;\xi} \\
[\![\langle \tau \rangle A]\!]_{\rho;\xi} &= \{s \mid \exists s', s.t.\ s \xrightarrow{\tau} s' \wedge s' \in [\![A]\!]_{\rho;\xi}\} \\
[\![\langle u!v \rangle A]\!]_{\rho;\xi} &= \{s \mid \exists s', s.t.\ s \xrightarrow{\bar{u}\rho v\rho} s' \wedge s' \in [\![A]\!]_{\rho;\xi}\} \\
[\![\langle u?(x) \rangle A]\!]_{\rho;\xi} &= \{s \mid \exists s', s.t.\ s \xrightarrow{u\rho(b)} s' \wedge s'[c/b] \in [\![A]\!]_{\rho[x \mapsto c];\xi} \\
&\qquad \text{for all } c \in \mathcal{N}\} \\
[\![\langle u!(x) \rangle A]\!]_{\rho;\xi} &= \{s \mid \exists s', s.t.\ s \xrightarrow{\bar{u}\rho(b)} s' \wedge s'[c/b] \in [\![A]\!]_{\rho[x \mapsto c];\xi} \\
&\qquad \text{for some } c \notin fn(s) \cup fn(A)\} \\
[\![\langle u?v \rangle A]\!]_{\rho;\xi} &= \{s \mid \exists s', s.t.\ s \xrightarrow{u\rho(b)} s' \wedge s'[v\rho/b] \in [\![A]\!]_{\rho;\xi}\} \\
[\![\Lambda(\tilde{u})]\!]_{\rho;\xi} &= [\![\Lambda]\!]_{\rho;\xi}(\rho(\tilde{u})) \\
[\![X]\!]_{\rho;\xi} &= \xi(X) \\
[\![(\tilde{x})A]\!]_{\rho;\xi} &= \lambda \tilde{y}.[\![A]\!]_{\rho[\tilde{x} \mapsto \tilde{y}];\xi} \\
[\![\nu X.\Lambda]\!]_{\rho;\xi} &= \sqcup\{F : \mathcal{N}^k \to \wp(\mathcal{G}) \mid F \sqsubseteq [\![\Lambda]\!]_{\rho;\xi[F/X]}\}
\end{aligned}
$$

Fig. 1. Interpretation of Formula

$[\![B]\!]_{\rho,\xi}$ for any ρ and ξ, which justifies our decision to identify α-equivalent formulas. Also, we can easily show the monotonicity of the semantics function Λ, since it is required that X occur positive in Λ. Thus, $\lambda f.[\![A]\!]_{\rho,\xi[X \mapsto f]}$ is a monotone functional over the complete lattice $(\{f : \mathcal{N}^k \to \wp(\mathcal{G})\}, \sqsubseteq)$. By Knaster-Tarski Theorem, we can draw the conclusion that $\nu X.\Lambda$ is the greatest fixpoint of $\lambda f.[\![A]\!]_{\rho,\xi[X \mapsto f]}$. The soundness of semantics can be obtained.

Now, we do some remarks on the choice of modality for the logical system. Generally speaking, there are two styles of syntax for the 'Hennessy-Milner logic' like systems for π-calculus. One is used by [14], the other is used by [4][6]. We follow the style of the former since from our point of view, it is clearer. However, the semantics is dramatically different. First, [14] lacks a modality for bound output, although the syntax $\langle \bar{a}(x) \rangle A$ exists in the logic system. It is not difficult to see that the semantics for $\langle \bar{a}(x) \rangle A$ does not coincide with the intuition very much. Second, the input modality in this paper coincides with the $\langle \bar{a}(x) \rangle^L$ in [14]. We don't introduce the corresponding modality for the other two 'input' modality because they can be rendered in our framework as

follows:

$$\langle a(b)\rangle A \stackrel{def}{=} \exists x.\langle a?x\rangle A \qquad\qquad \langle a(b)\rangle^E A \stackrel{def}{=} \forall x.\langle a?x\rangle A$$

The bound output modality needs more remarks. Note that our semantics for this modality coincides with Dam's though the syntax is different. To make this modality clearer, we consult to the fresh name quantification N and Dam's syntax a little. In fact,

$$\langle a!(x)\rangle A \stackrel{def}{=} \langle a\rangle \mathit{N}x.x \leftarrow A$$

We think reader who is familiar with N can easily understand this. We refer the reader to [2][3] for details. It is worth pointing out that as we mentioned in Section 1, such a quantification conveys difficulties when giving an interpretation though it is only implicit in our logic. The similar problems have been considered in [2]. As a remedy, [2] introduces the notion of *PSets* (Property sets). However, such a semantic device makes the semantics definition rather complex. Our solution is to make distinction between proposition and predicate, thus the possible interactions between recursion and first-order quantification can be solved. The advantage of our system lies in that the semantics of our logic is clearer and more concise. What's more, it is more favorable for model checking purpose. Also, it is worth pointing out that we need not introduce $\langle a!(b)\rangle$-like modality, since by the semantics, the choice of concrete name as the content of output action is immaterial, therefore, we use a variable instead of name.

Now, we set to establish some important results concerning the properties of logical formula, which is important for the model checking algorithm. Following [2], we use transposition as a useful tool to give some concise proof of properties concerning fresh names. Due to space restriction, most of detailed proofs are omitted. We refer the reader to the full version of this paper for more details. The following definition extends the notion of transposition to predicate.

Definition 3.2 Let θ be a transition. A function $f : \mathcal{N} \to \wp(\mathcal{G})$ is θ-preserving if $(f(n))\theta = f(n\theta)$ for any n. A valuation ξ is θ-preserving if $\xi(X)$ is θ-preserving for any X.

Lemma 3.3 *Given a transposition θ and a function $f : \mathcal{N} \to \wp(\mathcal{G})$, define $f^\theta : \mathcal{N} \to \wp(\mathcal{G})$ as $f^\theta(n) = f(n) \cup (f(n\theta))\theta$ for any n, then the following properties hold:*

(i) f^θ is θ-preserving.

(ii) If $f \sqsubseteq g$ and g is θ-preserving, then $f^\theta \sqsubseteq g$.

Proof. By the definition of transposition and θ-preserving, the proof is easy.□

Lemma 3.4 *Suppose ξ is θ-preserving, then the following properties hold:*

(i) $(\llbracket A \rrbracket_{\rho;\xi})\theta = \llbracket A\theta \rrbracket_{\rho;\xi}$.

(ii) $\llbracket \Lambda \rrbracket_{\rho;\xi}$ *is θ-preserving.*

Proof. By mutual induction on the structure of A and Λ.

□

According to the semantics of $\forall x.A$ and $\langle a?(x)\rangle A$, to check if $P \in \llbracket \forall x.A \rrbracket$ requires to instantiate x with every name. However, as the following lemma demonstrates, it is sufficient to consider only the free names of A plus one fresh name. This finite characterization will be exploited in the model checking algorithm.

Lemma 3.5 *Suppose $c \notin fn(s, A)$, then the following properties hold:*

(i) $s \in \llbracket \forall x.A \rrbracket_{\rho;\xi}$ *iff* $s \in \bigcap_{k \in fn(A) \cup \{c\}} \llbracket A \rrbracket_{\rho[x \mapsto k];\xi}$.

(ii) $s \in \llbracket \langle u?(x)\rangle A \rrbracket_{\rho;\xi}$ *iff there exists s' s.t.* $s \stackrel{\rho(u)(b)}{\rightarrow} s'$ *and* $s'[k/b] \in \llbracket A \rrbracket_{\rho[x \mapsto k];\xi}$ *for* $k \in fn(A) \cup fn(s) \cup \{c\}$.

The semantics definition of the $\langle a!(x)\rangle A$ is stated in 'existential' style, i.e. $s \models \langle a!(x)\rangle A$ if there is **some** fresh name c and s', such that $s \stackrel{a(b)}{\rightarrow} s'$ and $s'[c/b] \models A[c/x]$. We give such a definition because from our point of view, it may coincide with our intuition of bound output and restriction operator better. However, since c is not free in either s or A, this particular choice of c should not matter. That is, any other name d with $d \notin fn(s, A)$ should equally do. Thus indeed the semantics can also be characterized 'universally'.

Lemma 3.6 $s \in \llbracket \langle a!(x)\rangle A \rrbracket_{\rho;\xi}$ *iff there exists s' s.t.* $s \stackrel{\bar{a}(b)}{\rightarrow} s'$ *and* $s'[c/b] \in \llbracket A \rrbracket_{\rho[x \mapsto c];\xi}$ *for* ***every*** $c \notin fn(s, A)$.

4 Model Checking Algorithm

In this section, we devote to providing a model checking algorithm for our π-μ-logic. Based on the results in the last section, now the most challenging problem is to deal with the fixpoint operator. For propositional μ-calculus, many researchers have provided a lot of methods to solve this problem. We choose the so called local model checking algorithm since the global algorithm requires a prior construction of state space, which is impossible in our setting.

One of the notable features of such an algorithm is the mechanism used to keep track of unfolding fixpoint formula. There are two common, equivalent,

approaches to this problem: one is to introduce constants in which the name of fixpoints occurrences and to use a global rule of inference to detect previous unwinding, the other is to introduce *tags* into fixpoint formula, which remembers exactly the points of the model that have been visited before. As in [15], we adopt the latter of the methods and follow the framework due to Winskel [16], sometimes referred as tag set method. However, differing from [15], we lift it to the predicate case and in our algorithm, the tag sets will contain pairs $(\tilde{n}, s)$ of name vector and the node of STG. Formally, let $T = \{(\tilde{b_1}, s_1), \ldots, (\tilde{b_n}, s_n)\}$, where, $\tilde{n_i}$ $(1 \leq i \leq n)$ are vectors of the same length, say k and for $\forall i, j, i \neq j$, we have $\tilde{b}_i \neq \tilde{b}_j$. For any tag set T, we use λT to denote a function $\mathcal{N}^k \to \wp(\mathcal{G})$ defined as follows:

$$(\lambda T)(\tilde{b}) = \begin{cases} \bigcup_i \{s_i\} & \text{if } (\tilde{b}, s_i) \in T \\ \emptyset & \text{if o.w.} \end{cases}$$

Now, the fixpoint predicate $\nu X.\Lambda$ can be generalized to $\nu X.[T]\Lambda$, note that the X must have the same arity as T and the usage of T lies in recording what points of the model have been visited before thus is only a bookkeeping device. The definitions of $n(\nu X.[T]\Lambda)$, $fv(\nu X.[T]\Lambda)$ and $fpv(\nu X.[T]\Lambda)$ are the same as the corresponding definition for $\nu X.\Lambda$. Obviously, $\nu X.\Lambda$ can be covered as $\nu X.[]\Lambda$.

The denotation of $\nu X.[T]\Lambda$ is a simple extension for $[\![\nu X.\Lambda]\!]_{\rho;\xi}$ as follows:

$$[\![\nu X.[T]\Lambda]\!]_{\rho;\xi} = \sqcup\{F : \mathcal{N}^k \to \wp(\mathcal{G}) | F \sqsubseteq [\![\Lambda]\!]_{\rho;\xi[X \mapsto F]} \sqcup \lambda T\}$$

There now follows a technical lemma which is a generalization of the so called Reduction Lemma of [16], the essence of the tag set method.

Lemma 4.1 *(Reduction Lemma) Let $L = \mathcal{N}^k \to \wp(\mathcal{G})$ be a complete lattice and let ϕ be a monotone functional. Then for any $f \in L$,*

$$f \sqsubseteq \nu g.\phi(g) \quad \textit{iff} \quad f \sqsubseteq \phi(\nu g.(\phi(g) \sqcup f))$$

Proof. Straightforward generalization of the proof in [16]. □

Since X occurs positive in Λ, $\lambda f.[\![\Lambda]\!]_{\rho;\xi[X \mapsto f]} \sqcup \lambda T$ is a monotonic functional over $\mathcal{N}^k \to \wp(\mathcal{G})$, and $[\![\nu X.[T]\Lambda]\!]_{\rho;\xi}$ is its greatest fixpoint. So, using Lemma 4.1, the following lemma can be easily proved.

Lemma 4.2 *If $s \notin \lambda T(\tilde{b})$, then*

$$s \in [\![\nu X.[T]\Lambda]\!]_{\rho;\xi}(\tilde{b}) \quad \textit{iff} \quad s \in [\![\Lambda[\nu X.[T \cup \{(\tilde{b}, s)\}]\Lambda/X]]\!]_{\rho;\xi}(\tilde{b})$$

$$\mathrm{check}(m_\sigma, \phi) \stackrel{def}{=} Ev(\phi\sigma);$$

$$\mathrm{check}(m_\sigma, \neg A) \stackrel{def}{=} \neg\mathrm{check}(m_\sigma, A);$$

$$\mathrm{check}(m_\sigma, A \wedge B) \stackrel{def}{=} \mathrm{check}(m_\sigma, A) \wedge \mathrm{check}(m_\sigma, B);$$

$$\mathrm{check}(m_\sigma, \forall x.A) \stackrel{def}{=} \bigwedge_{b\in fn(m_\sigma,A)} \mathrm{check}(m_\sigma, A[b/x]) \wedge \mathrm{check}(m_\sigma, A[\mathrm{new}(fn(m_\sigma, A))/x]);$$

$$\mathrm{check}(m_\sigma, \langle\tau\rangle A) \stackrel{def}{=} \bigvee_{m \stackrel{\phi,\tau}{\mapsto} n} Ev(\phi\sigma) \wedge \mathrm{check}(n_\sigma, A);$$

$$\mathrm{check}(m_\sigma, \langle a?(x)\rangle A) \stackrel{def}{=} \bigvee_{m \stackrel{\phi,b(c)}{\mapsto} n} Ev(\phi\sigma \wedge [b\sigma = a]) \wedge \bigwedge_{d\in fn(m_\sigma,A)\cup\mathrm{new}(fn(m_\sigma,A))} \mathrm{check}(n_{\sigma[c\mapsto d]\restriction fn(n)}, A[d/x]);$$

$$\mathrm{check}(m_\sigma, \langle a!c\rangle A) \stackrel{def}{=} \bigvee_{m \stackrel{\phi,\bar{b}d}{\mapsto} n} Ev(\phi\sigma \wedge [a = b\sigma] \wedge [c = d\sigma]) \wedge \mathrm{check}(n_\sigma, A);$$

$$\mathrm{check}(m_\sigma, \langle a?b\rangle A) \stackrel{def}{=} \bigvee_{m \stackrel{\phi,c(d)}{\mapsto} n} Ev(\phi\sigma \wedge [a = c\sigma]) \wedge \mathrm{check}(n_{\sigma[d\mapsto b]\restriction fn(n)}, A);$$

$$\mathrm{check}(m_\sigma, \langle a!(x)\rangle A) \stackrel{def}{=} \bigvee_{m \stackrel{\phi,\bar{b}(c)}{\mapsto} n} Ev(\phi\sigma \wedge [b = a\sigma]) \wedge \mathrm{check}(n_{\sigma[c\mapsto \mathrm{new}(fn(m_\sigma,A))]}, A[\mathrm{new}(fn(m_\sigma, A))/x]);$$

$$\mathrm{check}(m_\sigma, (\nu X.[T]\Lambda)(\tilde{b})) = \begin{cases} true & \text{if } (\tilde{b}, m_\sigma) \in T \\ \mathrm{check}(m_\sigma, (\Lambda[\nu X.[T \cup \{(\tilde{b}, m_\sigma)\}]\Lambda/X])(\tilde{b})) & \text{o.w.} \end{cases}$$

Fig. 2. Local Model Checking Algorithm

Now, we present our algorithm as follows. The algorithm inputs a STG with the root n and substitution σ and a closed formula A. As usual, we denote m_σ as s. The algorithm returns *true* if $s = m_\sigma$ satisfies A, otherwise, it returns false. The pseudo-codes of the algorithm is presented in Fig.2. Recall that for a set of names $V \subset_{fin} \mathcal{N}$, function new($V$) returns the least name in $\mathcal{N}\backslash V$.

Now, we devote to proving the correctness of our algorithm. To establish the termination property of the algorithm, we need to bound on the number of names for model checking process, just as in [15] to bound on the free variables. First, we should point out that according to the rule for transferring process terms to STG, all bound names in a STG are different. And since we adopt the α-equivalence for formula, we can assume the bound name variables in a formula are also different. Now, we write $N_{\mathcal{G}}$ for the number of names

(including free and bound names) contained in the nodes of $\mathcal{G}$ and N_A for the number of names and name variables contained in A. Note that names in tag set of the formula does not be included, since it only contributes as a bookkeeping. The following lemma is important, by which we can conclude that provided that each term only appears once in each tag set (just as in our algorithm), the size of tag set is bounded since the STG we consider is finite.

Lemma 4.3 *For each recursive call of check, with caller parameter* (s, A) *and the callee parameter (s',A'),* $N_{s'} + N_{A'} \leq N_s + N_A$

Proof. By case analysis for each recursive call, the proof is routine. □

We now use this fact to give a well-founded ordering to formula. We write $A \ll_{\mathcal{G}} A'$ iff A' is not a fixpoint formula and A is a proper sub-formula of A', otherwise A is the form $(\Lambda[\nu X.[T \cup \{\tilde{b}, s\}]\Lambda/X])(\tilde{b})$ and A' is $(\nu X.[T]\Lambda)(\tilde{b})$ where $(\tilde{b}, s) \notin T$ and T contains only nodes from $\mathcal{G}$. We aim to show that the transitive closure $\ll_{\mathcal{G}}^{+}$ of this relation is a well-founded order whenever $\mathcal{G}$ is finite.

Lemma 4.4 *If* $\mathcal{G}$ *is finite, then* $\ll_{\mathcal{G}}^{+}$ *is a well-founded order.*

Proof. Similar to [15], Proposition 4. □

Theorem 4.5 *For any STG with root* r *and closed formula* A*, the following properties hold:*

(i) *check*(r, A) *terminates;*

(ii) *check*$(r, A) = true$ *iff* $r \in [\![A]\!]$.

Proof. (Sketch)

(i) Apply well-founded induction on $\ll_{\mathcal{G}}$, the termination is guaranteed.

(ii) Most cases follow closely the interpretation of formula. The cases for ϕ, $\neg A$ and $A \wedge B$, $\langle a?b\rangle A$ and $\langle a!u\rangle A$ are trivial. The correctness of case for $\forall x.A$ follows from Lemma 3.5(i). The case of $\langle a?(x)\rangle A$ follows Lemma 3.5(ii). The case of $\langle a!(x)\rangle A$ follows Lemma 3.6. For the fixpoint $(\nu X.[T]\Lambda)(\tilde{b})$, Lemma 4.2 applies.

The proof is completed. □

5 Conclusion

In this section, we conclude our work and discuss the related work. This paper deals with modal logic for mobile concurrent system and related model-checking algorithm. In this paper, we use symbolic transition graph to model

concurrent systems. A wide class of π calculus processes, that is, the finite-control processes can be represented as finite symbolic transition graph and such translation is rather simple.Thus our model has strong expressing power. A new version π-μ-Logic, an extension of the modal μ-calculus with boolean expressions over names, and primitives for name input and output is introduced as an appropriate modal logic for the π-calculus. Also our logical system can be seen as the extension for the modal logic in [14]. We give a concise semantics interpretation for our modal logic by making a distinction between proposition and predicate, thus the possible interactions between recursion and first-order quantification can be solved. Based on the above work, a local model checking algorithm for the logic is presented in this paper. We follow the well-known Winskel's tag set method to deal with fixpoint operator. As for the problem of name instantiating, our algorithm follows the 'on-the-fly' style, and systematically employs schematic names. The correctness of the algorithm is shown.

There is a sea of publications on model checking, but only a few address name-passing processes. The most related work on this paper may be [4][6], we have discussed his work in Introduction. There are also some work on value-passing process, e.g. [15] extends modal logic in [7] with fixpoints. The main concern of these papers is to formulate proof systems for deriving statements asserting whether a process satisfies a formula. Algorithm for model checking processes against such modal logics, was not payed much attention to. On the other direction, [11] deals with model checking value-passing CCS processes, by which some ideas of this paper is inspired. However, comparing the work on value-passing process, the modality of bound output conveys difficulties to both the semantics interpretation and model-checking algorithm, which is one of the main challenges on our work. We also should mention [2][3], the fresh name quantification concerning the bound output modality is studied there. We have compared our work with theirs in details in Section 3. We think our method, at least, is more favorable for model checking purpose. However, some useful tools, such as transposition, come from [2].

There are several directions for further research. How to improve efficiency of our algorithm, is an interesting problem. Also, we are investigating how to generate information diagnosis messages which will be useful in debugging a system when the answer returned by the algorithm is 'no'. Maybe we should apply symbolic technique in more depth.

References

[1] R.M.Amadio, M.Dam. Toward a Modal Theory of Types for the π Calculus. Proc. Formal Techniques in Real Time and Fault Tolerant Systems 96, Uppsala. LNCS 1135, Springer, 1996.

[2] L.Caires, L.Cardelli. A Spatical Logic for Concurrency (Part I). TACS'2001, Lecture Notes in Computer Science 2215, pp.1-30, Springer, 2001.

[3] L.Caires, L.Cardelli. A Spatical Logic for Concurrency (Part II). Proc. CONCUR'02, LNCS 2421, pp.209-225, Springer, 2002.

[4] M.Dam. Model Checking Mobile Processes. Information and Computation 129: 25-51, 1996.

[5] M.Dam. On the Decidability of Process Equivalences for the pi-Calculus. Theoretical Computer Science 183, pp. 215-228, 1997.

[6] M.Dam. Proof systems for π-Calculus Logics. To appear de Queiroz (ed.), "Logic for Concurrency and Synchronisation", Studies in Logic and Computation, Oxford Univ Press, 2003.

[7] M.Hennessy, X.Liu. A Modal Logic for Message Passing Processes. Acta Informatics, 32:375-393, 1995.

[8] M.Hennessy, R.Milner. Algebraic Laws for Non-determinism and Concurrency, Journal of ACM, Vol 32, pp. 137-161, 1985.

[9] Z.Li, H.Chen. Checking Strong/weak Bisimulation Equivalence and Observation Congruence for the π-calculus. In ICALP'98, Lecture Notes in Computer Science 1443, pp.707-718, Springer, 1998.

[10] H.Lin. Complete Inference Systems for Weak Bisimulation Equivalences in the π-calculus, TAPSOFT'95, Lecture Notes in Computer Science 915, pp.187-201, Springer, 1995.

[11] H.Lin. Model Checking Value-passing Processes. Proceedings of the 8th Asia-Pacific Software Engineering conference. Macau: IEEE Press, 2001, 3-10.

[12] R.Milner. Communication and Concurrency, Prentice Hall, 1989.

[13] R.Milner, J.Parrow, D.Walker. A Calculus of Mobile Process, part I/II. Journal of Information and Computation, 100:1-77, Sept.1992.

[14] R.Milner, J.Parrow, D.Walker. Modal Logics for Mobile Process. Theoretical Computer Science, 114:149-171, 1993.

[15] J.Rathke, M.Hennessy. Local Model Checking for Value-passing Process. Proceeding of the 9th International Conference on Theoretical Aspect of Computer Software, 1997.

[16] G.Winskel. A Note on Model Checking the Modal μ-calculus. Theoretical Computer Science 83:157-167, 1991.