A Process Logic for Distributed System Synthesis

Yoshinao Isobe Kazuhito Ohmaki
Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba, Ibaraki 305-8568, Japan
{isobe | ohmaki}@etl.go.jp

Abstract

In this paper, we define a process algebra DS@ to formally describe distributed systems and a process logic SP@ to formally describe their specifications. Then, we present a method to synthesize a distributed system (described in DS@) from given specifications (described in SP@). The main contribution of this paper is to show how to check the satisfiability of process logics in which concurrent behavior is distinct from interleaving behavior (i.e. considering true concurrency).

1. Introduction

The design of distributed systems is known to be a complex task, because the behavior of a distributed system results from interactions between concurrent processes of which the system consists. Our final purpose is to establish a method to automatically synthesize a distributed system from specifications. For this purpose, we need a formal framework to verify whether a distributed system satisfies a specification, or not.

Process algebras such as *CCS*[10] and *CSP*[2] are formal frameworks to express both specifications and concurrent systems, and to verify their behavioral equality. Langerak[7] presented algorithms to equivalently transform a sequential expression (a specification) to a concurrent expression (a system), by using a process algebra LOTOS[15]. However, requirements for the system behavior must be *completely* specified in LOTOS, because 'equality' is used between a specification and a system. Although extended process algebras[4, 8, 13] have been proposed to express *loose* specifications, it has not been discussed how to synthesize a concurrent process from sequential specifications.

Process logics such as $\mathcal{PL}$[10] (also called Hennessy-Milner logic) and μ-calculus[14] are logics with modal operators. In process logics, specifications can be flexibly expressed by disjunction operators $\vee$, and they can be refined by conjunction operators $\wedge$, step by step. Kimura et al.[6] presented an algorithm to synthesize a system (described in a process algebra CCS) from specifications (described in a process logic μ-calculus). However, since μ-calculus has no notion of concurrency, the algorithm may synthesize a sequential system (ex. $a.b+b.a$) which is observationally equal to the expected concurrent system (ex. $a|b$), where a, b are actions, '.' is a sequential operator, $+$ is a choice operator, and $|$ is a concurrent operator. Although the sequential system synthesized from logical specifications by the Kimura's algorithm, can be transformed into a concurrent system by the Langerak's algorithms[7], the concurrent system often contains more synchronizations than the logical specifications need, because the *medium* sequential system is synthesized without respect to concurrency. To synthesize an efficient concurrent system, concurrency should be considered in logical specifications.

As another approach using logics, Manna et al.[9] presented an algorithm to synthesize a graph from requirements described in Propositional Temporal Logic (PTL). However, concurrent requirements cannot be specified in PTL, in the same way as μ-calculus.

A number of process algebras which can express (true) concurrency have been proposed in [1, 3, 11], by considering locality or causality between actions. In addition, a process logic considering locality has been also given in [1]. The notion of true concurrency distinguishes concurrent behavior from interleaving behavior as follows: $a|b \neq a.b + b.a$. However, satisfiability of such process logics has not been discussed yet. The satisfiability check is necessary for synthesizing a distributed system from specifications.

In this paper, we define a true concurrent process algebra *DS*@ to describe distributed systems and a process logic *SP*@ to describe their specifications, in Section 2 and 3. Then, we present an algorithm to check the satisfiability of given specifications described in *SP*@, and a method to synthesize a distributed system described in *DS*@, in Section 4. Finally, we conclude this paper, and discuss abstract expressions in *SP*@, in Section 5.

2. Definition of distributed systems

In this section, we define a process algebra *DS@* to describe distributed systems. *DS@* is basically an extended CCS with *name operator* @ for naming each process, to specify where actions are performed. Such a notion is not new as shown in [5]. Our purpose is not to propose a new process algebra nor a process logic, but to present a synthesis method of distributed systems. Thus, we firstly need a *simple* formal framework suitable for the purpose. The basic results on *DS@* can be extended to other true concurrent process algebras.

At first, we assume that a set of *action names* $AN = \{a, b, \cdots\}$ and a set of *process names* $PN = \{p, q, \cdots\}$ are given, and they are ranged over by $\alpha, \beta, \cdots$ and $\psi, \varphi, \cdots$, respectively. And, subsets of PN are represented by $\Psi, \Phi, \cdots$. Each *action* is an action name with process names, and the set of actions Act is defined by

$$Act = \{\alpha\Psi : \alpha \in AN, \Psi \subseteq PN, 1 \leq \|\Psi\| \leq 2\},$$

where $\|\Psi\|$ is the number of elements in Ψ. Intuitively, $\alpha\{\psi\}$ is an action performed at the port named α in the process ψ, and $\alpha\{\psi, \varphi\}$ represents a synchronization between the two processes ψ and φ through the port α.

We assume that a set of *process-constants* $Cons^P$, ranged over by $A, B, \cdots$, is also given. Then, the set of *processes* Pr is the smallest set which contains the following expressions:

$$\begin{array}{rcl} \mathbf{0} & : & \text{Stop}, \\ A & : & \text{Recursion } (A \in Cons^P), \\ \alpha.P & : & \text{Prefix } (\alpha \in AN), \\ P + Q & : & \text{Summation}, \end{array}$$

where P, Q are already in Pr. The set Pr is ranged over by $P, Q, \cdots$. The set of *distributed systems* Ds is the smallest set which contains the following expressions:

$$\begin{array}{rcl} P@\psi & : & \text{Naming } (P \in Pr, \psi \in PN), \\ D|E & : & \text{Composition } (Pn(D) \cap Pn(E) = \emptyset), \\ \mathcal{E} \triangleright D & : & \text{Environment } (\mathcal{E} \subseteq Act), \end{array}$$

where D, E are already in Ds. The function $Pn : Ds \to 2^{PN}$ is defined as follows : $Pn(P@\psi) = \{\psi\}$, $Pn(D|E) = Pn(D) \cup Pn(E)$, and $Pn(\mathcal{E} \triangleright D) = Pn(D)$.

To avoid too many parentheses, these operators have the binding power: Prefix > Summation > Naming > Composition> Environment. The notation $D \equiv E$ is used to mean that D and E are syntactically identical.

Each operator is briefly explained as follows.

The process $\alpha.P$ can perform the action α, and then behaves like P. The process $P + Q$ behaves like either P or Q. The meaning of a process-constant is given by a defining equation. We assume that for every $A \in Cons^P$, there is a defining equation of the form $A \stackrel{\text{def}}{=} P$.

Name	Hypothesis	$\vdash$	Conclusion
Pre		$\vdash$	$\alpha.P \xrightarrow{\alpha\cdot} P$
$\mathbf{Sum}_1$	$P \xrightarrow{\alpha\cdot} P'$	$\vdash$	$P + Q \xrightarrow{\alpha\cdot} P'$
$\mathbf{Sum}_2$	$Q \xrightarrow{\alpha\cdot} Q'$	$\vdash$	$P + Q \xrightarrow{\alpha\cdot} Q'$
Rec	$A \stackrel{\text{def}}{=} P,\ P \xrightarrow{\alpha\cdot} P'$	$\vdash$	$A \xrightarrow{\alpha\cdot} P'$

Figure 1. The inference rules for $\dot{\to}$

Name	Hypothesis	$\vdash$	Conclusion
Name	$P \xrightarrow{\alpha\cdot} P'$	$\vdash$	$P@\psi \xrightarrow{\alpha\{\psi\}} P'@\psi$
$\mathbf{Com}_1$	$D \xrightarrow{\alpha\Psi} D'$	$\vdash$	$D\|E \xrightarrow{\alpha\Psi} D'\|E$
$\mathbf{Com}_2$	$E \xrightarrow{\alpha\Psi} E'$	$\vdash$	$D\|E \xrightarrow{\alpha\Psi} D\|E'$
$\mathbf{Com}_3$	$D \xrightarrow{\alpha\{\psi\}} D',\ E \xrightarrow{\alpha\{\varphi\}} E'$	$\vdash$	$D\|E \xrightarrow{\alpha\{\psi,\varphi\}} D'\|E'$
Env	$D \xrightarrow{\alpha\Psi} D', \alpha\Psi \in \mathcal{E}$	$\vdash$	$\mathcal{E} \triangleright D \xrightarrow{\alpha\Psi} \mathcal{E} \triangleright D'$

Figure 2. The inference rules for $\to$

$P@\psi$ names the process P the name ψ. The function Pn is used to uniquely assign a process name, and $Pn(D)$ represents the set of process names used in D. It is important that $(a.\mathbf{0}@p|b.\mathbf{0}@q)$ cannot be expanded to $(a@p.b@q.\mathbf{0} + b@q.a@p.\mathbf{0})$, because p and q are given to each process (not to each action), where $D|E$ represents a distributed system consisting of D and E.

An environment $\mathcal{E}$ is a set of feasible actions. Intuitively, it represents a network connection. For example, the environment $\{in\{p\}, out\{q\}, sync\{p, q\}\}$ shows that p and q can independently perform *in* and *out*, respectively, and can synchronize through *sync*.

The following function $env : Ds \to 2^{Act}$ is useful to estimate all the feasible actions:

$$\begin{array}{l} env(P@\psi) = \{\alpha\{\psi\} : \alpha \in AN\}, \\ env(D|E) = \{\alpha\{\psi, \varphi\} : \alpha\{\psi\} \in env(D), \alpha\{\varphi\} \in env(E)\} \\ \qquad\qquad \cup env(D) \cup env(E), \\ env(\mathcal{E} \triangleright D) = \{\alpha\Psi \in \mathcal{E} : \Psi \subseteq Pn(D)\} \cap env(D). \end{array}$$

And the function $pn : 2^{Act} \to 2^{PN}$ is often used for extracting process names from an environment, and it is defined as : $pn(\mathcal{E}) = \{\psi : \exists\alpha\Psi.\ \alpha\Psi \in \mathcal{E}, \psi \in \Psi\}$.

The semantics of processes and distributed systems is given by the labelled transition systems $\langle Pr, AN, \dot{\to}\rangle$ and $\langle Ds, Act, \to\rangle$, respectively, where the transitions $\dot{\to}$ and $\to$ are the smallest sets satisfying the inference rules in Figure 1 and Figure 2, respectively.

The rule $\mathbf{Com}_3$ restricts the number of synchronous actions to 2, although *DS@* is easily extended with n-synchronizations. This means that $\alpha\{\psi, \varphi\}$ cannot synchronize with the other actions. Thus, $\alpha\{\psi, \varphi\}$ is uncontrollable, and it corresponds to an *internal action*. The synchronous name α can be abstracted by defining an equation such that $\alpha\Psi \simeq \alpha'\Psi$ if $\|\Psi\| = 2$.

3. Definition of specifications

In this section, a process logic *SP@* is defined to describe specifications. *SP@* is an extended $\mathcal{PL}$ with process names, recursion, and a cost operator. Intuitively, the cost operator indicates that the system synthesized from $S_1 \vee S_2$ should satisfy which specification S_1 or S_2, if both specifications are satisfiable.

We assume that a set of *specification-constants* (also called *Constants*) *Cons*, ranged over by $A, B, \cdots$, is given. Then, the set of *specifications Sp* is the smallest set which contains the following expressions:

$$\begin{array}{rcl} \mathtt{tt} & : & \text{True}, \\ \mathtt{ff} & : & \text{False}, \\ A & : & \text{Recursion } (A \in \mathit{Cons}), \\ \langle\alpha\Psi\rangle S & : & \text{Possibility } (\alpha\Psi \in \mathit{Act}), \\ [\alpha\Psi]S & : & \text{Necessity } (\alpha\Psi \in \mathit{Act}), \\ S \wedge T & : & \text{Conjunction}, \\ S \vee T & : & \text{Disjunction}, \\ r{::}S & : & \text{Cost } (r : \text{a positive real number}), \end{array}$$

where S, T are already in *Sp*. The set *Sp* is ranged over by $S, T, \cdots$. To avoid too many parentheses, these operators have binding power such that: { Possibility, Necessity, Cost } > Conjunction > Disjunction.

The possibility $\langle\alpha\Psi\rangle S$ requires that the action $\alpha\Psi$ can be performed, and then S can be satisfied after $\alpha\Psi$. And the necessity $[\alpha\Psi]S$ requires that if the action $\alpha\Psi$ can be performed, then S is always satisfied after $\alpha\Psi$.

In the same way as process-constants, a Constant is a specification whose meaning is given by a defining equation. We assume that for every Constant $A \in \mathit{Cons}$, there is a defining equation of the form $A \stackrel{\text{def}}{=} S$.

The cost operator is mainly used for expressing communication costs. For example, the specification $5{::}\langle c\{p_1, p_2\}\rangle S \vee 3{::}\langle c\{p_1, p_3\}\rangle S$ indicates that the communication cost between the processes p_1 and p_2 is higher than the cost between p_1 and p_3. To synthesize a system from such disjunctive specifications, a low cost specification should be selected as far as possible.

Next, an extended labelled transition system is defined to give the semantics of specifications.

Definition 3.1 *A* requirement labelled transition system (*RLTS*) *is a tuple* $\langle St, Lb, \mapsto, \longrightarrow_{\langle\rangle}, \longrightarrow_{[]}\rangle$, *where St is a set of* states, *Lb is a set of* labels, $\mapsto \subseteq St \times St$ *is a* disjunctive transition, $\longrightarrow_{\langle\rangle} \subseteq St \times Lb \times St$ *is a* possible transition, $\longrightarrow_{[]} \subseteq St \times Lb \times St$ *is a* necessary transition. *We use* ξ *to range over the set* $\{\langle\rangle, []\}$, *and write* $\longrightarrow_\xi$ *for* $\longrightarrow_{\langle\rangle}$ *or* $\longrightarrow_{[]}$. *We often write* $s \mapsto s'$ *for* $(s, s') \in \mapsto$ *and* $s \stackrel{e}{\longrightarrow}_\xi s'$ *for* $(s, e, s') \in \longrightarrow_\xi$. ∎

In the RLTS, each transition represents a requirement. Intuitively, a state $s \in St$ is satisfied, if for some s' such that $s \mapsto s'$, every transition $s' \stackrel{e}{\longrightarrow}_\xi s''$ is satisfied. A possible transition $s \stackrel{e}{\longrightarrow}_{\langle\rangle} s'$ requires that e can be performed and then s' can be satisfied, and a necessary transition $s \stackrel{e}{\longrightarrow}_{[]} s'$ requires that if e can be performed then s' is always satisfied after e. Note that a state s such that $s \not\mapsto$ is never satisfied, and s such that for some s', $s \mapsto s' \not\longrightarrow_\xi$ is always satisfied.

Name	Hypothesis	$\vdash$	Conclusion
$\mathbf{True}_\vee$		$\vdash$	$\mathtt{tt} \mapsto \mathtt{tt}$
$\mathbf{Pos}_\vee$		$\vdash$	$\langle\alpha\Psi\rangle S \mapsto \langle\alpha\Psi\rangle S$
$\mathbf{Nec}_\vee$		$\vdash$	$[\alpha\Psi]S \mapsto [\alpha\Psi]S$
$\mathbf{Con}_\vee$	$S \mapsto S_0, T \mapsto T_0$	$\vdash$	$S \wedge T \mapsto S_0 \wedge T_0$
$\mathbf{Dis}_{1\vee}$	$S \mapsto S_0$	$\vdash$	$S \vee T \mapsto S_0$
$\mathbf{Dis}_{2\vee}$	$T \mapsto T_0$	$\vdash$	$S \vee T \mapsto T_0$
$\mathbf{Rec}_\vee$	$A \stackrel{\text{def}}{=} S,\ S \mapsto S_0$	$\vdash$	$A \mapsto S_0$
$\mathbf{Cos}_\vee$	$S \mapsto S_0$	$\vdash$	$r{::}S \mapsto r{::}S_0$

Figure 3. The inference rules of $\mapsto$

Name	Hypothesis	$\vdash$	Conclusion
Pos		$\vdash$	$\langle\alpha\Psi\rangle S \stackrel{\alpha\Psi}{\longrightarrow}_{\langle\rangle} S$
Nec		$\vdash$	$[\alpha\Psi]S \stackrel{\alpha\Psi}{\longrightarrow}_{[]} S$
$\mathbf{Con}_1$	$S_0 \stackrel{\alpha\Psi}{\longrightarrow}_\xi S'$	$\vdash$	$S_0 \wedge T_0 \stackrel{\alpha\Psi}{\longrightarrow}_\xi S'$
$\mathbf{Con}_2$	$T_0 \stackrel{\alpha\Psi}{\longrightarrow}_\xi T'$	$\vdash$	$S_0 \wedge T_0 \stackrel{\alpha\Psi}{\longrightarrow}_\xi T'$
Cos	$S_0 \stackrel{\alpha\Psi}{\longrightarrow}_\xi S'$	$\vdash$	$r{::}S_0 \stackrel{\alpha\Psi}{\longrightarrow}_\xi S'$

Figure 4. The inference rules of $\longrightarrow_\xi$

The semantics of specifications is given by the RLTS $\langle \mathit{Sp}, \mathit{Act}, \mapsto, \longrightarrow_{\langle\rangle}, \longrightarrow_{[]}\rangle$, where the transitions $\mapsto$ and $\rightarrow_\xi$ are the smallest sets satisfying the inference rules in Figure 3 and Figure 4, respectively.

It can be proven that if $S \mapsto S' \mapsto S''$ then $S' \equiv S''$. This means that it is not necessary to successively perform disjunctive transitions twice. The set of *undisjunctive* specifications just after a disjunctive transition is denoted by Sp_0, and it is ranged over by $S_0, T_0, \cdots$. Thus, $\mathit{Sp}_0 = \{S_0 : \exists S \in \mathit{Sp}.\ S \mapsto S_0\}$. Note that undisjunctive specifications may become disjunctive specifications after a possible (or a necessary) transition.

Then, the satisfaction is defined as follows.

Definition 3.2 *A set* $\mathcal{R} \subseteq Ds \times Sp$ *is a* satisfaction set, *if* $(D, S) \in \mathcal{R}$ *implies that for some* S_0, $S \mapsto S_0$ *and the following* $(i), (ii)$ *hold for every* $\alpha\Psi$ *and* S',

(i) *if* $S_0 \stackrel{\alpha\Psi}{\longrightarrow}_{\langle\rangle} S'$, *then* $\exists D'.\ D \stackrel{\alpha\Psi}{\longrightarrow} D'$ *and* $(D', S') \in \mathcal{R}$,
(ii) *if* $S_0 \stackrel{\alpha\Psi}{\longrightarrow}_{[]} S'$ *and* $D \stackrel{\alpha\Psi}{\longrightarrow} D'$, *then* $(D', S') \in \mathcal{R}$.

Then, if $(D, S) \in \mathcal{R}$ *for some satisfaction set* $\mathcal{R}$, *then* D satisfies S, *written* $D \models S$. ∎

Definition 3.2 requires that there is S_0 such that $S \mapsto S_0$ and $(i), (ii)$ are satisfied. The idea of the disjunctive transition was proposed in [12]. And, (i) requires that D can perform the action $\alpha\Psi$, and it can satisfy S' after $\alpha\Psi$, and (ii) requires that if D can perform the action $\alpha\Psi$, then it always satisfies S' after $\alpha\Psi$. Proposition 3.1 shows the property of $\models$.

Proposition 3.1

(1) $D \models \mathtt{tt},\ D \not\models \mathtt{ff}$
(2) $D \models \langle\alpha\Psi\rangle S \ \Leftrightarrow\ \exists D'.\ (D \xrightarrow{\alpha\Psi} D',\ D' \models S)$
(3) $D \models [\alpha\Psi] S \ \Leftrightarrow\ \forall D'.\ (D \xrightarrow{\alpha\Psi} D' \Rightarrow D' \models S)$
(4) $D \models S \wedge T \ \Leftrightarrow\ D \models S,\ D \models T$
(5) $D \models S \vee T \ \Leftrightarrow\ D \models S \ or\ D \models T$
(6) $D \models r::S \ \Leftrightarrow\ D \models S$
(7) $D \models A \ \Leftrightarrow\ \exists S.\ (D \models S,\ A \stackrel{\text{def}}{=} S)$

∎

4. Satisfiability and synthesis

A number of incomplete specifications (in which requirements have not been uniquely fixed) are sometimes given to a system instead of its complete specification, because many designers work on the same system design in parallel. Such design method decreases responsibility of each designer, but it raises two important issues: *consistency check* of the incomplete specifications and *synthesis* of a system to satisfy them.

A process logic can express incomplete specifications by disjunction operators $\vee$ and so on. And the consistency of specifications $S_1, \cdots, S_n$ can be checked by the satisfiability of $S_1 \wedge \cdots \wedge S_n$.

In addition, not only behaviors but environments are often required for distributed systems. Thus, it is important to check whether distributed systems to satisfy given behaviors and environments exist, or not. This conditional satisfiability is defined as follows.

Definition 4.1 *Let $\mathcal{E} \subseteq Act$. S is* $\mathcal{E}$-satisfiable *if and only if for some D, $D \models S$ and $env(D) = \mathcal{E}$.* ∎

In this section, we present an algorithm to check the $\mathcal{E}$-satisfiability of a specification described in *SP*@, and to synthesize a distributed system described in *DS*@.

4.1. Problem

If process names are removed, the satisfiability of *SP*@ can be checked by imposing each necessary requirement $[a]S$ on possible requirements $\langle a\rangle T$ for each state, like $\langle a\rangle(S \wedge T)$. For example, the requirement of the following S_1 is equal to the requirement of S_1'.

$$S_1 \equiv \langle a\rangle\langle b\rangle\mathtt{tt} \wedge [a]\mathtt{ff}$$
$$S_1' \equiv \langle a\rangle(\langle b\rangle\mathtt{tt} \wedge \mathtt{ff}) \wedge [a]\mathtt{ff}$$

Thus, S_1 is not satisfiable, because S_1' requires the false $\mathtt{ff}$ after $\langle a\rangle$. On the other hand, the following S_2 is satisfiable,

$$S_2 \equiv \langle a\rangle\langle b\rangle\mathtt{tt} \wedge [b]\mathtt{ff}$$

because the action b is forbidden by $[b]\mathtt{ff}$ *until* some action is performed. Thus, b can be performed after a.

However, if process names are attached, then the satisfiability check is not so easy, because independency of processes must be considered. For example, although the differences between S_2 and the following S_3 are only process names, S_3 is not satisfiable,

$$S_3 \equiv \langle a\{p\}\rangle\langle b\{q\}\rangle\mathtt{tt} \wedge [b\{q\}]\mathtt{ff}$$

because the action b is forbidden by $[b\{q\}]\mathtt{ff}$ until some action is performed by the process q. This means that the process q cannot know the state-changes of the other process p without communications. Thus, the process q cannot perform b even after p has performed a.

In general, a necessary requirement for a process may be imposed on a possible requirement for the process, *passing* requirements for the other processes. For example, the following $[a\{p\}]\mathtt{ff}$ may be imposed on $\langle a\{p\}\rangle\mathtt{tt}$, passing $\langle b\{q\}\rangle, \cdots, [c\{q\}], \cdots$.

$$[a\{p\}]\mathtt{ff} \wedge \langle b\{q\}\rangle(\cdots([c\{q\}](\cdots\langle a\{p\}\rangle\mathtt{tt}) \vee S_1)) \wedge S_2$$

Therefore, every such possible requirement should be checked at the beginning. However, it is difficult, because requirements are not fixed by disjunctions. Furthermore, it is impossible to fix every disjunction at the beginning, because of recursions. Thus, we *stepwise* check the satisfiability by a *trial and error method*.

4.2. Preliminary

Our algorithm assigns specifications in an array whose dimension is decided by process names, in order to clarify independency of processes.

At first, for each $\Psi \subseteq PN$, the set of *pointers* Pnt_Ψ ranged over by $u, v, \cdots$, the set of *arrays* Ary_Ψ ranged over by $\rho, \sigma, \cdots$, and the set of *array-transitions* Trn_Ψ ranged over by $r, s, \cdots$, are defined as follows:

$$Pnt_\Psi = \begin{cases} \{\emptyset\} & (\Psi=\emptyset), \\ \{\{(\psi; i)\} : i \in \mathcal{I}\} & (\Psi=\{\psi\}), \\ \{u \cup v : u \in Pnt_\Phi, v \in Pnt_\Theta\} & (\Psi=\Phi\cup\Theta, \Phi\cap\Theta\neq\emptyset), \end{cases}$$
$$Ary_\Psi = \{\rho : \rho \subseteq \{(u, S) : u \in Pnt_\Psi, S \in Sp\}\},$$
$$Trn_\Psi = \{r : r \subseteq \mathcal{I} \times AN \times PN \times \mathcal{I}\},$$

where $\mathcal{I}$ is the set of integers. Intuitively, each point $\{(\psi_n; i_n) : n \in N\}$ represents a state of a distributed system in which the *state-ID* of process ψ_n is i_n, and each element (u, S) in an array represents that a state of a distributed system pointed by u satisfies S.

The set Ary^0_Ψ, ranged over by $\rho_0, \sigma_0, \cdots$, is an important subset of Ary_Ψ, and it is given as follows:

$$Ary^0_\Psi = \{\rho_0 : \rho_0 \subseteq \{(u, S_0) : u \in Pnt_\Psi, S_0 \in Sp_0\}\}.$$

The pointer $u[v] \in Pnt_\Psi$ obtained by replacing a part of $u \in Pnt_\Psi$ with $v \in Pnt_\Phi$ is defined as follows:

$$u[v] = \{(\psi; i) \in u : \psi \notin \Phi\} \cup \{(\psi; i) \in v : \psi \in \Psi\}.$$

For example, $u[v] = \{(p_1; 1), (p_2; 4), (p_3; 3)\}$, where $u = \{(p_1; 1), (p_2; 2), (p_3; 3)\}$ and $v = \{(p_2; 4), (p_5; 5)\}$. And the state-ID of the process ψ in a pointer u is denoted by $u(\psi)$. For example, $\{(p; 3), (q; 4)\}(p) = 3$.

Then, the important notion to check $\mathcal{E}$-satisfiability is defined as follows.

Definition 4.2 *Let $\mathcal{E} \subseteq Act$, $\rho_0 \in Ary^0_{pn(\mathcal{E})}$, and $r \in Trn_{pn(\mathcal{E})}$. A pair (ρ_0, r) is* $\mathcal{E}$-closed *if and only if for every $(u, S_0) \in \rho_0$ and for every $\alpha\Psi$ and S',*

(1) *if $S_0 \xrightarrow{\alpha\Psi}_{\Diamond} S'$, then $\exists v \in Pnt_\Psi.\ \alpha\Psi \in \mathcal{E}$, $(\forall \psi \in \Psi.\ (u(\psi), \alpha, \psi, v(\psi)) \in r)$, $\exists S'_0.\ S' \mapsto S'_0$, and $(u[v], S'_0) \in \rho_0$,*

(2) *if $S_0 \xrightarrow{\alpha\Psi}_{[\!]} S', v \in Pnt_\Psi, \alpha\Psi \in \mathcal{E}$, and $(\forall \psi \in \Psi.\ (u(\psi), \alpha, \psi, v(\psi)) \in r)$, then $\exists S'_0.\ S' \mapsto S'_0$, and $(u[v], S'_0) \in \rho_0$.*

∎

For example, (ary, trn) is (net)-closed, where

$$\begin{aligned} ary &= \{((1,1), S_{01}), ((2,1), S_{02}), ((4,1), S_{03}), \\ &\quad ((1,3), 1{::}\langle c\{p,q\}\rangle S_{03}), ((2,3), \langle e\{q\}\rangle S_2)\}, \\ trn &= \{(1, a, p, 2), (1, c, p, 4), (4, d, p, 1), \\ &\quad (1, b, q, 3), (3, c, q, 1), (3, e, q, 1)\} \\ net &= \{a\{p\}, b\{q\}, c\{p, q\}, d\{p\}, e\{q\}\} \end{aligned}$$

where each (i, j) in ary is an abbreviation of the pointer $\{(p; i), (q; j)\}$, and each S_i is defined as follows:

$$\begin{aligned} &S_1 \stackrel{\text{def}}{=} S_{01} \equiv \langle a\{p\}\rangle S_2 \wedge [b\{q\}]S_3 \wedge [d\{p\}]\mathtt{ff}, \\ &S_2 \stackrel{\text{def}}{=} S_{02} \equiv \langle b\{q\}\rangle\langle e\{q\}\rangle S_2, \\ &S_3 \stackrel{\text{def}}{=} S_{03} \vee 1{::}\langle c\{p,q\}\rangle S_{03}, \qquad S_{03} \equiv \langle d\{p\}\rangle S_1. \end{aligned}$$

Figure 5 shows the requirement graph of S_1 and the environment net. The array ary and the array-transition trn are illustrated as shown in Figure 6.

The following Propositions 4.1 and 4.2 show the necessary and sufficient condition for $\mathcal{E}$-satisfiable. Thus, a specification S is $\mathcal{E}$-satisfiable if and only if for some $\mathcal{E}$-closed pair (ρ_0, r), ρ_0 contains S_0 such that $S \mapsto S_0$.

Proposition 4.1 *If $D \models S$, then for some $env(D)$-closed pair (ρ_0, r), for some $(u, S_0) \in \rho_0$, $S \mapsto S_0$.* ∎

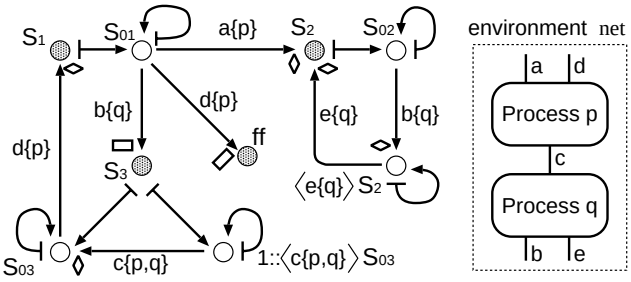

Figure 5. The requirements S_1 and net

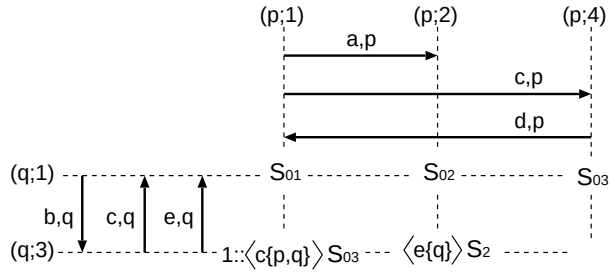

Figure 6. The (net)-closed pair (ary, trn)

Proposition 4.2 *Assume that (ρ_0, r) is $\mathcal{E}$-closed. If $(u, S_0) \in \rho_0$ and $S \mapsto S_0$, then $\mathbf{Ds}^r_{\mathcal{E}}(u) \models S$, where the distributed system $\mathbf{Ds}^r_{\mathcal{E}}(u)$ is defined as follows*:

$$\begin{aligned} &\mathbf{Ds}^r_{\mathcal{E}}(u) \equiv \mathcal{E} \triangleright \textstyle\prod\{\mathbf{Pr}^r_\psi(i)@\psi : (\psi; i) \in u\}, \\ &\mathbf{Pr}^r_\psi(i) \stackrel{\text{def}}{=} \textstyle\sum\{\alpha.\mathbf{Pr}^r_\psi(i') : (i, \alpha, \psi, i') \in r\}, \end{aligned}$$

where $\prod\{D_i : 1 \le i \le n\}$ and $\sum\{P_i : 1 \le i \le n\}$ represent $D_1|\cdots|D_n$ and $P_1 + \cdots + P_n$, respectively. ∎

Furthermore, Proposition 4.2 gives a method to synthesize a distributed system to satisfy a specification. For example, the following distributed system $\mathbf{Ds}^{trn}_{net}(1, 1)$ synthesized from the previous (net)-closed pair (ary, trn) satisfies S_1 by Proposition 4.2.

$$\begin{aligned} \mathbf{Ds}^{trn}_{net}(1,1) &\equiv net \triangleright (\mathbf{Pr}^{trn}_p(1)@p | \mathbf{Pr}^{trn}_q(1)@q) \\ \mathbf{Pr}^{trn}_p(1) &\stackrel{\text{def}}{=} a.\mathbf{0} + c.d.\mathbf{Pr}^{trn}_p(1) \\ \mathbf{Pr}^{trn}_q(1) &\stackrel{\text{def}}{=} b.(c.\mathbf{Pr}^{trn}_q(1) + e.\mathbf{Pr}^{trn}_q(1)) \end{aligned}$$

4.3. Algorithm

For each $\mathcal{E} \subseteq Act$, the algorithm $\mathbf{Sat}^{\mathcal{E}}$ is presented in Figure 7. This algorithm checks whether distributed systems (whose environment is $\mathcal{E}$) to satisfy a specification exist, or not. The input is a specification to be checked, and the output is a tuple (b, ρ_0, r), where $b \in \{\mathtt{tt}, \mathtt{ff}\}$, $\rho_0 \in Ary^0_{pn(\mathcal{E})}$, and $r \in Trn_{pn(\mathcal{E})}$. The meaning of the output is shown in Theorem 4.3.

$\mathbf{Sat}^{\mathcal{E}}(S) = \mathbf{Dis}^{\mathcal{E}}(\emptyset, \emptyset, \sigma)$, where $\sigma = \{\{((\psi; 1) : \psi \in pn(\mathcal{E})\}, S)\}$.

$\mathbf{Dis}^{\mathcal{E}}(\rho_0, r, \sigma) = (b, \rho_0', r')$, where
- if $\sigma = \emptyset$, then $(b, \rho_0', r') := (\mathtt{tt}, \rho_0, r)$, ($\sharp_1$)
- else if $\exists (u, S) \in \sigma.\ S \not\mapsto$, then $(b, \rho_0', r') := (\mathtt{ff}, \rho_0, r)$, ($\sharp_2$)
- else $(b, \rho_0', r') := \mathbf{Rec}^{\mathcal{E}}(\rho_0, r, \sigma_0 - \rho_0)$, where
  - $\sigma_0 \in \mho_0 := \{\sigma_0' : (\forall (u, S) \in \sigma.\ \exists S_0.\ S \mapsto S_0, (u, S_0) \in \sigma_0'), (\forall (u, S_0) \in \sigma_0'.\ \exists S.\ S \mapsto S_0, (u, S) \in \sigma)\}$, ($\sharp_3$)
  - $(check(\mathbf{Rec}^{\mathcal{E}}(\rho_0, r, \sigma_0 - \rho_0)) = \mathtt{tt}$ or $(\forall \sigma_0' \in \mho_0.\ \sigma_0' \ll_D \sigma_0))$, ($\sharp_4$)
  - $(\forall \sigma_0' \in \mho_0$ such that $\sigma_0' \ll_D \sigma_0.\ check(\mathbf{Rec}^{\mathcal{E}}(\rho_0, r, \sigma_0' - \rho_0)) = \mathtt{ff})$. ($\sharp_5$)

$\mathbf{Rec}^{\mathcal{E}}(\rho_0, r, \sigma_0) = \mathbf{Pos}^{\mathcal{E}}(g(\rho_0), g(r), g(\sigma_0) - g(\rho_0))$, where
- $g \in G := \{g' : \forall (\psi, i).\ (g'(\psi, i) = i$ or $\exists (u, S_0) \in \sigma_0.\ \exists u'.\ (u', S_0) \in \rho_0 \cup \sigma_0, u(\psi) = i, u'(\psi) = g'(\psi, i)\}$, ($\sharp_6$)
- $(check(\mathbf{Pos}^{\mathcal{E}}(g(\rho_0), g(r), g(\sigma_0) - g(\rho_0))) = \mathtt{tt}$ or $(\forall g' \in G.\ g' \ll_R g))$, ($\sharp_7$)
- $(\forall g' \in G$ such that $g' \ll_R g.\ check(\mathbf{Pos}^{\mathcal{E}}(g'(\rho_0), g'(r), g'(\sigma_0) - g'(\rho_0))) = \mathtt{ff})$. ($\sharp_8$)

$\mathbf{Pos}^{\mathcal{E}}(\rho_0, r, \sigma_0) = (b, \rho_0', r')$, where
- if $\{\alpha\Psi : \exists (u, S_0) \in \sigma_0.\ \exists S'.\ S_0 \xrightarrow{\alpha\Psi}_{\Diamond} S'\} - \mathcal{E} \neq \emptyset$, then $(b, \rho_0', r') = (\mathtt{ff}, \rho_0, r)$, ($\sharp_9$)
- else $(b, \rho_0', r') = \mathbf{Nec}^{\mathcal{E}}(\rho_0 \cup \sigma_0, r \cup r'', \sigma)$, where
  - $r'' := \{(u(\psi), \alpha, \psi, v(\psi)) : \exists S'.\ \exists \Psi.\ (u, \alpha\Psi, v, S') \in new, \psi \in \Psi\}$, ($\sharp_{10}$)
  - $\sigma := \{(u[v], S') : \exists \alpha\Psi.\ (u, \alpha\Psi, v, S') \in new\}$, ($\sharp_{11}$)
  - $new := \{(u, \alpha\Psi, v, S') : \exists S_0.\ (u, S_0) \in \sigma_0, S_0 \xrightarrow{\alpha\Psi}_{\Diamond} S', v = \{(\psi; newid(\rho_0 \cup \sigma_0, u, \alpha\Psi, S')) : \psi \in \Psi\}\}$. ($\sharp_{12}$)

$\mathbf{Nec}^{\mathcal{E}}(\rho_0, r, \sigma) = \mathbf{Dis}^{\mathcal{E}}(\rho_0, r, \sigma \cup \sigma'')$, where
- $\sigma' := \{(u[v], S') : \exists S_0.\ \exists \alpha\Psi \in \mathcal{E}.\ (u, S_0) \in \rho_0, S_0 \xrightarrow{\alpha\Psi}_{[]} S', v \in Pnt_\Psi, (\forall \psi \in \Psi.\ (u(\psi), \alpha, \psi, v(\psi)) \in r)\}$, ($\sharp_{13}$)
- $\sigma'' := \{(u, S') \in \sigma' : \forall S_0'$ such that $S' \mapsto S_0'.\ (u, S_0') \notin \rho_0\}$. ($\sharp_{14}$)

Figure 7. The algorithm $\mathbf{Sat}^{\mathcal{E}}$ for $\mathcal{E}$-satisfiability check

Theorem 4.3 *Assume that* $\mathbf{Sat}^{\mathcal{E}}(S)$ *terminates.*

(1) $check(\mathbf{Sat}^{\mathcal{E}}(S)) = \mathtt{tt}$ *if and only if* S *is* $\mathcal{E}$*-satisfiable, where for each* (b, ρ_0, r), $check(b, \rho_0, r) = b$.

(2) *If* $\mathbf{Sat}^{\mathcal{E}}(S) = (\mathtt{tt}, \rho_0, r)$, *then* (ρ_0, r) *is* $\mathcal{E}$*-closed and for some* $(u, S_0) \in \rho_0$, $S \mapsto S_0$. ∎

By Propositions 4.2 and Theorem 4.3, if S is $\mathcal{E}$-satisfiable, then a distributed system can be synthesized. However, we have not proven the termination of the algorithm $\mathbf{Sat}^{\mathcal{E}}(S)$ yet, even if S has finite states (i.e. S can contain recursive requirements, if the number of reachable states is finite), although we speculate that $\mathbf{Sat}^{\mathcal{E}}(S)$ terminates. The proof is not so trivial by disjunctions, recursions, and independency of processes. If a specification S contains no disjunction and has finite states (may contain recursions), then we have already proven that $\mathbf{Sat}^{\mathcal{E}}(S)$ terminates.

4.3.1. Outline of the algorithm $\mathbf{Sat}^{\mathcal{E}}$. The algorithm $\mathbf{Sat}^{\mathcal{E}}$ consists of four sub-algorithms $\mathbf{Dis}^{\mathcal{E}}$, $\mathbf{Rec}^{\mathcal{E}}$, $\mathbf{Pos}^{\mathcal{E}}$, and $\mathbf{Nec}^{\mathcal{E}}$. The meaning of the input tuple $(\rho_0, r, \sigma$(or $\sigma_0))$ to the sub-algorithms is as follows: The array ρ_0 contains specifications whose possible transitions have been checked. The array σ (or σ_0) contains specifications whose disjunctive transitions (or possible transitions) have not been checked yet. And, r is the array-transition produced by possible transitions of specifications in ρ_0. The meaning of the output is the same as one of $\mathbf{Sat}^{\mathcal{E}}$. Then, each sub-algorithm is explained, where ($\sharp_n$) points a line in Figure 7.

$\mathbf{Dis}^{\mathcal{E}}$ returns $\mathtt{tt}$, if there is no specification to be checked ($\sharp_1$), and it returns $\mathtt{ff}$, if some specification has no disjunctive transition ($\sharp_2$). Otherwise, it selects the *least* set σ_0 (with respect to the order $\ll_D$) of $\mathcal{E}$-satisfiable undisjunctive specifications from the set $\mho_0$ ($\sharp_{3,4,5}$). If every σ_0' in $\mho_0$ is not $\mathcal{E}$-satisfiable, then the greatest one is selected ($\sharp_4$). The order $\ll_D$ is explained with the order $\ll_R$ in $\mathbf{Rec}^{\mathcal{E}}$ at the end of this subsubsection.

$\mathbf{Rec}^{\mathcal{E}}$ attempts to fold the array $(\rho_0 \cup \sigma_0)$ for creating recursive processes. The algorithm $\mathbf{Sat}^{\mathcal{E}}$ cannot terminate for recursive specifications without $\mathbf{Rec}^{\mathcal{E}}$. A function $g : PN \times \mathcal{I} \to \mathcal{I}$ is a *renumbering function.*

The function renumbering the state i of process ψ to j is denoted by $(\psi; i \to j)$. This function is extended over pointers, arrays, and array-transitions as follows:

$$g(u) = \{(\psi; g(\psi, i)) : (\psi; i) \in u\},$$
$$g(\rho) = \{(g(u), S) : (u, S) \in \rho\},$$
$$g(r) = \{(g(\psi, i), \alpha, \psi, g(\psi, i')) : (i, \alpha, \psi, i') \in r\}.$$

The set G collects renumbering functions g' such that if $i \neq g'(\psi, i)$, then $(*)$ *both the states* i *and* $g'(\psi, i)$ *of the process* ψ *satisfy the same specification* ($\sharp_6$). Thus, it creates a recursive process. However, *the condition* $(*)$ is not a sufficient condition to identify the two states of the process ψ, because a disjunctive specification is satisfied by two or more different processes. Therefore, G always contains the identical function id such that for every (ψ, i), $id(\psi, i) = i$.

$\mathbf{Pos}^{\mathcal{E}}$ firstly checks whether actions forbidden by the environment $\mathcal{E}$ are required, or not ($\sharp_9$). If such an action exists, then $\mathbf{Pos}^{\mathcal{E}}$ returns $\mathtt{ff}$. Otherwise, it adds new transitions into r, according to possible transitions of specifications contained in σ_0 ($\sharp_{10,12}$). And it sets the destinations of the possible transitions in σ, to satisfy the condition (1) in Definition 4.2 ($\sharp_{11}$). The function *newid* assigns a new identical integer which is not used in $\rho_0 \cup \sigma_0$, for each $(u, \alpha\Psi, S')$.

$\mathbf{Nec}^{\mathcal{E}}$ searches all the necessary transitions induced by transitions in r, and it adds the destinations of the necessary transitions in σ, to satisfy the condition (2) in Definition 4.2 ($\sharp_{13}$), where the destinations which have already checked are removed from σ ($\sharp_{14}$).

The orders $\ll_D$ and $\ll_R$ in $\mathbf{Dis}^{\mathcal{E}}$ and $\mathbf{Rec}^{\mathcal{E}}$ are used for selecting an element from the sets $\mho_0$ and G, respectively ($\sharp_{5,8}$). It is important to note that Theorem 4.3 does not depend on how to select an element by the orders $\ll_D$ and $\ll_R$. Therefore, it is not necessary to carefully define the orders. However, the following points should be considered.

- The order $\ll_D$ should be defined such that a low cost specification is selected. For example, the cost of each specification $S_0 \in Sp_0$ is defined as:

$$Cos(\mathtt{tt}) = Cos(\langle\alpha\Psi\rangle S) = Cos([\alpha\Psi]S) = 0,$$
$$Cos(S_0 \wedge T_0) = Cos(S_0) + Cos(T_0),$$
$$Cos(r{::}S_0) = r + Cos(S_0).$$

- The order $\ll_R$ should be defined such that a renumbering function which changes many different integers into an integer is selected, in order to create recursive processes and terminate $\mathbf{Sat}^{\mathcal{E}}$.

4.3.2. Demonstration. To demonstrate the algorithm $\mathbf{Sat}^{\mathcal{E}}$, the example in Subsection 4.2 is used again. By applying $\mathbf{Sat}^{net}$ to the specification S_1 of Figure 5, the tuple $(\mathtt{tt}, \rho'_{05}, r'_5)$ is returned as shown in Figure 8, where (ρ'_{05}, r'_5) is the same as (ary, trn) given in Subsection 4.2.

In Figure 8, the steps 6, 7, and 11 are important. In the step 6, the element $((1,3), S_3)$ is induced by $S_{01} \xrightarrow{b\{q\}}_{[\![}\ S_3$ and $S_{02} \xrightarrow{b\{q\}}_{\langle\rangle}\ \langle e\{q\}\rangle S_2$. This shows that requirements for the process q are preserved between S_{01} and S_{02}, although there is a requirement $\langle a\{p\}\rangle$ for the process p between S_{01} and S_{02}.

In the step 7, $1{::}\langle c\{p,q\}\rangle S_{03}$ is selected from S_3, although the cost 1 is higher than the cost 0 of S_{03}, because if S_{03} is selected in this step, then it is inconsistent with $[d\{p\}]\mathtt{ff}$. The synchronization $c\{p,q\}$ is needed for satisfiable.

In the step 11, the state-ID 5 of the process q is changed into 1 by g_1, because $((2,5), S_{02}) \in \sigma_{05}$ and $((2,1), S_{02}) \in \rho_{04}$. By the renumbering, a recursion is created, and the element $((2,5), S_{02})$ in σ_{05} is removed.

5. Conclusion and discussion

We have presented an algorithm $\mathbf{Sat}^{\mathcal{E}}$ to check the $\mathcal{E}$-satisfiability of a process logic *SP*@, in which concurrent behavior is distinct from interleaving behavior. Although the termination of $\mathbf{Sat}^{\mathcal{E}}$ has not been proven yet, $\mathbf{Sat}^{\mathcal{E}}$ is useful for synthesizing a distributed system described in a process algebra *DS*@ from specifications described in *SP*@.

SP@ has no abstract synchronous name like τ of CCS, because an abstract name makes the algorithm $\mathbf{Sat}^{\mathcal{E}}$ be (somewhat) more complex, and such name can be expressed by short notations in *SP*@ as follows:

$$\langle\tau\rangle S \equiv \bigvee\{\langle\alpha\Psi\rangle S : \alpha \in AN, \Psi \subseteq PN, \|\Psi\| = 2\},$$
$$[\tau]S \equiv \bigwedge\{[\alpha\Psi]S : \alpha \in AN, \Psi \subseteq PN, \|\Psi\| = 2\},$$

where $\bigvee\{S_i : 1 \leq i \leq n\}$ and $\bigwedge\{S_i : 1 \leq i \leq n\}$ represent $S_1 \vee \cdots \vee S_n$ and $S_1 \wedge \cdots \wedge S_n$, respectively. We can remove every concrete synchronous name from *DS*@ and *SP*@ by using $\langle\tau\rangle S$, $[\tau]S$, and the equation $\simeq$ over synchronous actions given at the end of Section 2.

It seems to be difficult for designers to directly describe specifications in *SP*@, because *SP*@ is very primitive. Then, short notations are helpful for designers. For example, synchronizations (communications) can be abstracted by the following notations:

$$\langle\!\langle\alpha\Psi\rangle\!\rangle S \equiv \langle\alpha\Psi\rangle S \vee \langle\tau\rangle\langle\alpha\Psi\rangle S,$$
$$[[\alpha\Psi]]S \equiv [\alpha\Psi]S \wedge [\tau][\alpha\Psi]S.$$

Intuitively, a synchronization is allowed before the action $\alpha\Psi$ by $\langle\!\langle\alpha\Psi\rangle\!\rangle S$, and the necessity for the action $\alpha\Psi$ is available after a synchronization by $[[\alpha\Psi]]S$. More abstract operators can be defined by combination of

1	$\mathbf{Sat}^{net}(S_1) = \mathbf{Dis}^{net}(\emptyset, \emptyset, \sigma_1)$	$\sigma_1 := \{((1,1), S_1)\}$
2	$= \mathbf{Rec}^{net}(\emptyset, \emptyset, \sigma_{01}) = \mathbf{Pos}^{net}(\emptyset, \emptyset, \sigma_{01})$	$\sigma_{01} := \{((1,1), S_{01})\}$
3	$= \mathbf{Nec}^{net}(\sigma_{01}, r_1, \sigma_2) = \mathbf{Dis}^{net}(\sigma_{01}, r_1, \sigma_2)$	$r_1 = \{(1, a, p, 2)\}, \sigma_2 := \{((2,1), S_2)\}$
4	$= \mathbf{Rec}^{net}(\sigma_{01}, r_1, \sigma_{02}) = \mathbf{Pos}^{net}(\sigma_{01}, r_1, \sigma_{02})$	$\sigma_{02} := \{((2,1), S_{02})\}$
5	$= \mathbf{Nec}^{net}(\rho_{02}, r_2, \sigma_3)$	$\rho_{02} := \sigma_{01} \cup \sigma_{02}, r_2 := r_1 \cup \{(1, b, q, 3)\}, \sigma_3 := \{((2,3), \langle e\{q\}\rangle S_2)\}$
6	$= \mathbf{Dis}^{net}(\rho_{02}, r_2, \sigma_4)$	$\sigma_4 := \sigma_3 \cup \{((1,3), S_3)\}$
7	$= \mathbf{Rec}^{net}(\rho_{02}, r_2, \sigma_{04}) = \mathbf{Pos}^{net}(\rho_{02}, r_2, \sigma_{04})$	$\sigma_{04} := \sigma_3 \cup \{((1,3), 1::\langle c\{p,q\}\rangle S_{03})\}$
8	$= \mathbf{Nec}^{net}(\rho_{04}, r_4, \sigma_5)$	$\rho_{04} := \rho_{02} \cup \sigma_{04}, r_4 := r_2 \cup \{(1, c, p, 4), (3, c, q, 4), (3, e, q, 5)\},$
9	$= \mathbf{Dis}^{net}(\rho_{04}, r_4, \sigma_5)$	$\sigma_5 := \{((4,4), S_{03}), ((2,5), S_2)\}$
10	$= \mathbf{Rec}^{net}(\rho_{04}, r_4, \sigma_{05})$	$\sigma_{05} := \{((4,4), S_{03}), ((2,5), S_{02})\}, g_1 := (q; 5 \to 1)$
11	$= \mathbf{Pos}^{net}(\rho_{04}, r'_4, \sigma'_{05})$	$r'_4 := g_1(r_4), \sigma'_{05} := g_1(\sigma_{05}) - g_1(\rho_{04}) = \{((4,4), S_{03})\}$
12	$= \mathbf{Nec}^{net}(\rho_{05}, r_5, \sigma_6) = \mathbf{Dis}^{net}(\rho_{05}, r_5, \sigma_6)$	$\rho_{05} := \rho_{04} \cup \sigma'_{05}, r_5 := r'_4 \cup \{(4, d, p, 6)\}, \sigma_6 := \{((6,4), S_1)\}$
13	$= \mathbf{Rec}^{net}(\rho_{05}, r_5, \sigma_{06})$	$\sigma_{06} := \{((6,4), S_{01})\}, g_2 := (p; 6 \to 1)(q; 4 \to 1)$
14	$= \mathbf{Pos}^{net}(\rho'_{05}, r'_5, \emptyset) = \mathbf{Nec}^{net}(\rho'_{05}, r'_5, \emptyset)$	$\rho'_{05} := g_2(\rho_{05}), r'_5 := g_2(r_5), g_2(\sigma_{06}) - \rho'_{05} = \emptyset$
15	$= \mathbf{Dis}^{net}(\rho'_{05}, r'_5, \emptyset) = (\mathtt{tt}, \rho'_{05}, r'_5)$	

Figure 8. The application of $\mathbf{Sat}^{net}$ to the specification S_1

basic operators. By these short notations, designers can describe *week* specifications without respect to synchronizations. If synchronizations are needed, then they are automatically inserted by the algorithm $\mathbf{Sat}^{\mathcal{E}}$

Acknowledgments

The authors wish to express our gratitude to Dr. Izumi Takeuti for helpful discussions.

References

[1] G.Boudol, I.Castellani, M. Hennessy, and A.Kiehn: Observing localities, *Theoretical Computer Science*, Vol.114, pp.31-61, 1993.

[2] C.A.R.Hoare :*Communicating Sequential Processes*, Prentice-Hall, 1985.

[3] P.Degano and C.Priami : Causality for Mobile Processes, ICALP'95, LNCS 944, Springer-Verlag, pp.660 - 671, 1995.

[4] Y.Isobe, Y.Sato and K.Ohmaki : Least Fixpoint and Greatest Fixpoint in a Process Algebra with Conjunction and Disjunction, *IEICE Trans. on Fundamentals,* Vol.E83-A, No.3, pp.401-411, 2000.

[5] P.Krishnan : Distributed CCS, CONCUR'91, LNCS 527, Springer-Verlag, pp.393-407, 1991.

[6] S.Kimura, A.Togashi and N.Shiratori : Synthesis Algorithm for Recursive Processes by μ-calculus, Algorithmic Learning Theory, LNCS 872, pp.379-394, 1994.

[7] R.Langerak : Decomposition of functionality : a correctness preserving LOTOS transformation, P-STV X, pp.229-242, 1990.

[8] K.G.Larsen, B.Steffen, and C.Weise : A Constraint Oriented Proof Methodology Based on Modal Transition Systems, TACAS'95, LNCS 1019, pp.17-40, 1995.

[9] Z.Manna and P.Wolper : Synthesis of Communicating Processes from Temporal Logic Specifications, *ACM Trans. on Programming Languages and Systems*, Vol.6, No.1, pp.67-93, 1984.

[10] R.Milner : *Communication and Concurrency*, Prentice-Hall, 1989.

[11] U.Montanari and D.Yankelevich : A Parametric Approach to Localities, ICALP'92, LNCS 623, Springer-Verlag, pp.617 - 628, 1992.

[12] M.W.A.Steen, H.Bowman, J.Derrick, and E.A. Boiten : Disjunction of LOTOS specification, FORTE X/PSTV XVII, pp.177-192, 1997.

[13] M.W.A.Steen : Consistency and Composition of Process Specifications (Chap.5). Ph.D Thesis, University of Kent at Canterbury, 1998.

[14] C.Stirling : An Introduction to Modal and Temporal Logics for CCS, Concurrency: Theory, Language, and Architecture, LNCS 491, Springer-Verlag, pp.2-20, 1989.

[15] ISO 8807: Information Processing Systems–Open System Interconnection–LOTOS–A formal description technique based on the temporal ordering of observational behavior, 1989.