J Comb Optim (2015) 29:308–329
DOI 10.1007/s10878-014-9779-0

A structural transformation from p-π to MSVL

Ling Luo · Zhenhua Duan · Cong Tian · Xiaobing Wang

Published online: 7 August 2014

Abstract This paper presents a structural transformation approach from p-π processes to MSVL programs. To this end, channel and communication primitives are firstly defined in MSVL. Further, based on these definitions, a mapping function $\mathcal{F}$ which transforms bounded p-π processes into MSVL programs is formalized. Moreover, the soundness of the transformation is proved. By the transformation, p-π can provide a mechanism to model, simulate and verify concurrent time-dependent systems by means of the techniques of MSVL. Finally, a case study is given to illustrate how the transformation can be used in practice.

Keywords Process algebra · Temporal logic · π-calculus · MSVL · Verification

1 Introduction

Process algebra is a useful formalism for describing and analyzing communicating and concurrent systems. Following CCS (Milner 1980), CSP (Hoare 1985) and ACP (Bergstra and Klop 1985), many kinds of variations of process algebras have been proposed for different purposes. Especially, with the development of new network computing technologies, π-calculus (Milner 1999; Milner et al. 1992; Sangiorgi and

L. Luo · Z. Duan (✉)· C. Tian · X. Wang
ICTT and ISN Lab, Xidian University, Xi'an 710071, People's Republic of China
e-mail: zhhduan@mail.xidian.edu.cn

L. Luo
e-mail: lingluo@stu.xidian.edu.cn

C. Tian
e-mail: ctian@mail.xidian.edu.cn

X. Wang
e-mail: xbwang@mail.xidian.edu.cn

Walker 2002) is such a variation widely used in practice to ensure the correctness of mobile systems. However, in π-calculus, the time duration of an action and the residence time of a system at a state are not taken into account. Thus, it is not convenient for modeling time-dependent systems. p-π (Luo and Duan 2012) is an extension of π-calculus for specifying time-dependent aspects of mobile systems. Although p-π provides plenty of algebraic approaches to specify and analyze concurrent time-dependent systems, it is not easy to verify these systems automatically by means of model checking.

On the other hand, projection temporal logic (PTL) (Duan et al. 1994; Duan 1996, 2006) is a useful formalism for describing sequences of transitions between states in reactive systems and offers abundant mechanisms for verifications. MSVL (Duan et al. 2008b; Yang et al 2010; Yang and Duan 2008), an executable subset of PTL, is a modeling, simulation and verification language. The minimal model, operational and axiomatic semantics have been systematically investigated, which provide prerequisite and convenience for automatically formal verification. Moreover, an interpreter for MSVL has been developed and it can work in three modes: modeling, simulation and verification. Since MSVL is useful for modeling and verifying concurrent time-dependent systems, it is convenient to express p-π.

Therefore, in this paper, we are motivated to investigate how to transform p-π processes into MSVL programs, so that we can take advantage of techniques of MSVL to verify properties of time-dependent systems modeled by p-π. To this end, we present a mapping function $\mathcal{F}$ which provides a structural translation from p-π processes into corresponding MSVL programs. The contribution of this paper is three-fold: (1) we improved the mapping function $\mathcal{F}$ and definitions of channel and communication primitives given in an abstract conference paper (Luo and Duan 2013), and proved all theorems; (2) we investigated the consistency between interleaving and true concurrency; (3) we proved the soundness of the transformation.

The rest of the paper is organized as follows. The syntax and semantics of p-π is briefly introduced in Sect. 2. In Sect. 3, first the syntax and semantics of MSVL are introduced; further, the channel and communication primitives are formally defined. Section 4 is devoted to the structural transformation from p-π processes to MSVL programs. In Sect. 5, the soundness of the transformation is proved. A case study is given in Sect. 6. Finally, conclusions are drawn in Sect. 7.

2 p-π

p-π is an extension of π-calculus to include interval action prefixes. Let $\mathcal{N}$ be an infinite countable set of names, PR a countable set of propositions, and $\mathbb{N}$ the set of non-negative integers. The syntax of p-π processes is given below.

$$\pi ::= x(y) \mid \overline{x}\langle y\rangle \mid \tau \mid I_p \mid skip \mid \varepsilon$$
$$P ::= 0 \mid \pi \cdot P \mid P_1 + P_2 \mid P_1 | P_2 \mid [a_1 = a_2]P \mid \nu a\ P \mid A\langle a_1, \ldots, a_n\rangle$$

where x, y, a, and a_i (i is an integer from 1 to $n \in \mathbb{N}$) are names ranging over $\mathcal{N}$.

The action prefix π falls into two categories: instantaneous action prefix and interval one. An instantaneous action prefix represents either sending (denoted by $\overline{x}\langle y\rangle$) or

receiving (denoted by $x(y)$) a message, or executing a silent τ or empty transition ε. An interval action prefix represents either I_p=$\{p_1 \wedge skip, \ldots, p_k \wedge skip\}$ or $skip$. Here $p_i \in PR(1 \leq i \leq k$ and $i \in \mathbb{N})$, and $skip$ is a special proposition indicating a time unit. For convenience, we use $\tilde{p}_i$ to represent $p_i \wedge skip$ and $\overline{x}$ (or x) to denote $\overline{x}\langle\rangle$ (or $x()$). Note that $\overline{x}\langle\rangle$ (or $x()$) means an action prefix without a message.

A process (or process expression) can be an empty process 0, an action prefix guarded process $\pi \cdot P$, summation of two processes $P_1 + P_2$, parallel composition of two processes $P_1|P_2$, a match structure $[a_1{=}a_2]P$, a restriction structure $\nu a\ P$, or a process instance $A\langle a_1, \ldots, a_n\rangle$. $[a_1{=}a_2]P$ indicates that P is executed if $a_1{=}a_2$. $\nu a\ P$ behaves as P except that the communication on the bound name a is forbidden. A in $A\langle a_1, \ldots, a_n\rangle$ is a process identifier defined by $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A$ and $A\langle a_1, \ldots, a_n\rangle$=$\{\overrightarrow{a}\,/\,\overrightarrow{x}\}P_A$ where $\overrightarrow{a}$ and $\overrightarrow{x}$ are the vectors of $a_1, \ldots, a_n$ and $x_1, \ldots, x_n$, respectively.The set of names in a process $n(P)$ consists of bound names $bn(P)$ and free names $fn(P)$.Names, say a, appearing in (a) or $\nu a\ P$ are bound names. Others are free names. The abbreviations about $skip$ and $await$ are defined as follows.

Skip_0 $\quad skip^0 \stackrel{\text{def}}{=} \varepsilon$

Skip_n $\quad skip^n \stackrel{\text{def}}{=} skip \cdot skip^{n-1} (n \geq 1)$

Await_d $\quad await(P) \stackrel{\text{def}}{=} \varepsilon \cdot P + skip \cdot P + \cdots + skip^n \cdot P (n \geq 0)$

where $n \in \mathbb{N}$. The derived process $await(P)$ is used to realize synchronous communication. Here P in $await(P)$ is a sending (or receiving) prefix guarded process.

Definition 1 The structural congruence over the set of p-π processes $\mathcal{P}$ is defined by the sixteen equations below.

S1 $P|0 \equiv P$ S2 $\nu x P \equiv \nu y\{y/x\}P$ if $y \notin fn(P)$

S3 $P + Q \equiv Q + P$ S4 $x(y) \cdot P \equiv x(z) \cdot \{z/y\}P$ if $z \notin fn((y)P)$

S5 $P|Q \equiv Q|P$ S6 $(P|Q)|R \equiv P|(Q|R)$

S7 $\nu x\ 0 \equiv 0$ S8 $\nu x\ (P|Q) \equiv P|\nu x\ Q$, if $x \notin fn(P)$

S9 $\nu x y\ P \equiv \nu y x\ P$ S10 $[x = y]P \equiv 0$, if $x \neq y$

S11 $\varepsilon \cdot P \equiv P$ S12 $skip \cdot P + skip \cdot Q \equiv skip \cdot (P + Q)$

S13 $P + 0 \equiv P$ S14 $A\langle a_1, \ldots, a_n\rangle \equiv \{\overrightarrow{a}\,/\,\overrightarrow{x}\}P_A$ if $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A$

S15 $\nu x\ \overline{x}\langle y\rangle \cdot P \equiv 0$ S16 $P + (Q + R) \equiv (P + Q) + R$

The execution of p-π processes consists of two stages: one for instantaneous action prefixes and the other one for interval action prefixes and the execution of an interval action prefix is assumed to take an interval with one time unit. For the operational rules, the observable actions is given as follows:

$$\pi_o ::= \bar{x}\langle y\rangle \mid x(z) \mid \bar{x}(z) \mid \tau \mid I_p \mid skip$$
$$\pi_c ::= \bar{x}\langle y\rangle \mid x(z) \mid \bar{x}(z) \mid \tau$$
$$\pi_t ::= I_p \mid skip$$

$$\text{Tau:}\frac{}{\tau \cdot P \xrightarrow{\tau} P} \qquad \text{In:}\frac{}{x(z)\cdot P \xrightarrow{x(w)} \{w/z\}P}(w \notin fn((z)P))$$

$$\text{Out:}\frac{}{\bar{x}\langle y\rangle \cdot P \xrightarrow{\bar{x}\langle y\rangle} P} \qquad \text{Par:}\frac{P_1 \xrightarrow{\pi_c} P_1'}{P_1|P_2 \xrightarrow{\pi_c} P_1'|P_2}(bn(\pi_c)\cap fn(P_2)=\emptyset)$$

$$\text{Sum:}\frac{P_1 \xrightarrow{\pi_c} P_1'}{P_1{+}P_2 \xrightarrow{\pi_c} P_1'} \qquad \text{Close:}\frac{P_1 \xrightarrow{\bar{x}(y)} P_1'\ \ P_2 \xrightarrow{x(y)} P_2'}{P_1|P_2 \xrightarrow{\tau} \nu y\,(P_1'|P_2')}$$

$$\text{Com:}\frac{P_1 \xrightarrow{\bar{x}\langle y\rangle} P_1'\ \ P_2 \xrightarrow{x(z)} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|\{y/z\}P_2'} \qquad \text{Mat:}\frac{P \xrightarrow{\pi_o} P'}{[a_1=a_2]P \xrightarrow{\pi_o} P'}(a_1=a_2)$$

$$\text{Open:}\frac{P \xrightarrow{\bar{x}\langle y\rangle} P'}{\nu y\ P \xrightarrow{\bar{x}(z)} \{z/y\}P'}\ (x\neq y, z\notin fn(\nu y P'))$$

$$\text{Res:}\frac{P \xrightarrow{\pi_o} P'}{\nu x P \xrightarrow{\pi_o} \nu x P'}(x\notin n(\pi_o)) \qquad \text{Ide:}\frac{\{\overrightarrow{a}/\overrightarrow{x}\}P_A \xrightarrow{\pi_o} P'}{A\langle\overrightarrow{a}\rangle \xrightarrow{\pi_o} P'}(A(\overrightarrow{x}) \stackrel{\text{def}}{=} P_A)$$

$$\text{Act}_t\text{:}\frac{}{\pi_t \cdot P \xrightarrow{\pi_t} P} \qquad \text{Sum}_{t_1}\text{:}\frac{P_1 \xrightarrow{skip} P_1'\ \ P_2 \xrightarrow{skip} P_2'}{P_1{+}P_2 \xrightarrow{skip} P_1'{+}P_2'}$$

$$\text{Act}_\varepsilon\text{:}\frac{P \xrightarrow{\pi_c} P'}{\varepsilon \cdot P \xrightarrow{\pi_c} P'} \qquad \text{Await:}\frac{\varepsilon\cdot P{+}skip\cdot await(P) \xrightarrow{\pi_o} P'}{await(P) \xrightarrow{\pi_o} P'}$$

$$\text{Com}_{idle}\text{:}\frac{P_1 \xrightarrow{\pi_t} P_1'\ \ P_2\equiv 0}{P_1|P_2 \xrightarrow{\pi_t} P_1'} \qquad \text{Sum}_{t_2}\text{:}\frac{P_1 \xrightarrow{\pi_t} P_1'}{P_1{+}P_2 \xrightarrow{\pi_t} P_1'}(P_1{+}P_2 \not\xrightarrow{skip})$$

$$\text{Com}_t\text{:}\frac{P_1 \xrightarrow{\pi_{t_1}} P_1'\ \ P_2 \xrightarrow{\pi_{t_2}} P_2'}{P_1|P_2 \xrightarrow{\pi_{t_1}\cup\pi_{t_2}} P_1'|P_2'}(\pi_{t_1}\cap\pi_{t_2}{-}skip=\emptyset, P_1|P_2 \not\xrightarrow{\tau})$$

Fig. 1 Operational semantics of p-π

where x and y are free names in $fn(\pi_o)$ or $fn(\pi_c)$. z is a bound name in $bn(\pi_o)$ or $bn(\pi_c)$. Here, $\overline{x}(z)$ is defined by $\nu z\ \overline{x}\langle z\rangle$.

As in [12], a transition in the form of $P \xrightarrow{\pi_o} Q$ in p-π means that P can evolve into Q after performing action π_o. In Fig. 1, *late* operational semantics of p-π is defined.

Definition 2 For any p-π process P, the LTS (Labeled Transition System) of P is a pair $(\mathcal{P}, \mathcal{T})$, in which $\mathcal{P}$ is the set of p-π processes and $\mathcal{T}$ is the set of transitions. $\mathcal{P}$ and $\mathcal{T}$ are defined as follows:

1. $P \in \mathcal{P}$;
2. For all $Q \in \mathcal{P}$, if $Q \xrightarrow{\pi_o} Q'$, then $Q' \in \mathcal{P}$, $(Q, \pi_o, Q') \in \mathcal{T}$. For convenience, we use LTS_P to denote the LTS produced from process P.

3 Framed temporal logic programming language MSVL

Our underlying logic is Projection Temporal Logic (PTL) and MSVL is an executable subset of PTL. MSVL consists of expressions and statements. The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$e ::= c|v|v_p|\&x|*v_p|\bigcirc x|\ominus x|e_0 op e_1 \qquad (op ::= +|-|*|/)$$
$$b ::= \mathsf{true}|\mathsf{false}|\neg b|b_0 \wedge b_1|e_0 = e_1|e_0 < e_1$$

where c is a constant, v a data variable, v_p a pointer variable and x denotes a data variable or a pointer variable. & is a pointer reference and $*$ a pointer dereference. $\bigcirc$ means *next* and $\ominus$ *previous*. A dynamic variable x is said to be framed in program prog if $\mathsf{frame}(x)$ or $\mathsf{lbf}(x)$ is contained in prog.

A framed program in MSVL can be formalized by the sixteen elementary statements below.$p_1, \ldots, p_m$, p and q are general framed programs.

Termination :	ε	Basic assignment :	$x = e$
Pointer assignment :	$*v_p = e$	State frame :	$\mathsf{lbf}(x)$
Interval frame :	$\mathsf{frame}(x)$	Conjuction :	$p \wedge q$
Selection :	$p \vee q$	Next statement :	$\bigcirc p$
Always statement :	$\Box p$	Conditional statement:	`if` b `then` p `else` q
Existential quantification :	$\exists x : p(x)$	Sequential statement :	p **;** q
Parallel :	$p \parallel q$	While statement :	`while` b `do` p
Synchronized communication :	$\mathsf{await}(b)$	Projection :	$(p_1, \ldots, p_m)$ prj q

ε is the termination statement. The basic assignment $x = e$ means that the value of variable x is equal to the value of expression e. Similarly, $*v_p = e$ is the assignment associated with the pointer. The unit assignment $x := e$ is defined as $x := e \stackrel{\text{def}}{=} skip \wedge \bigcirc x = e$. The *next* statement $\bigcirc p$ means that p holds at the immediate successor state. $\Box p$ implies that p holds in all states from now on. The sequential statement p **;** q signifies that p holds from the current state until some point in future at which it terminates and q will start executing from that point.

The state frame $\mathsf{lbf}(x)$ and interval frame $\mathsf{frame}(x)$ utilizes framing technique. The framing operator is denoted by frame and $\mathsf{frame}(x)$ means that *variable x always keeps its old value over an interval if no assignment to x is encountered*. To give its definition, a new assignment is formalized as $x \Leftarrow e \stackrel{\text{def}}{=} x = e \wedge p_x$ where p_x is an atomic proposition connected with variable x and cannot be used for other purposes. Then the assignment flag is formalized as: $\mathsf{af}(x) \stackrel{\text{def}}{=} p_x$. Thus, $\mathsf{lbf}(x)$ and $\mathsf{frame}(x)$ can be defined as $\mathsf{lbf}(x) \stackrel{\text{def}}{=} \neg\mathsf{af}(x) \rightarrow \exists\, a : (\ominus x = a \wedge x = a)$ and $\mathsf{frame}(x) \stackrel{\text{def}}{=} \Box(\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x))$ respectively, where a is a static variable.

The conditional statement `if` b `then` p `else` q first evaluates b; if b is true, then p is executed, otherwise q is executed. The iteration `while` b `do` p allows process p to be repeatedly executed a finite (or infinite) number of times as long as the condition b is satisfied at the beginning of each execution. The selection statement $p \vee q$ represents that p or q will be executed. The conjunction statements $p \wedge q$ declares that the processes p and q are executed concurrently sharing all the states and variables during the mutual execution. The parallel construction $p \parallel q$, shows another concurrent computation manner. The distinguished difference between $p \parallel q$ and $p \wedge q$ is that the former allows both p and q to be able to specify their own intervals while the latter does not. E.g., $\mathsf{len}(2) \parallel \mathsf{len}(3)$ holds but $\mathsf{len}(2) \wedge \mathsf{len}(3)$ is obviously

false. The statement await(b) is used to synchronize communication between parallel processes.

Projection $(p_1, \ldots, p_m)$ prj q claims that q is executed in parallel with $p_1, \ldots, p_m$ over an interval obtained by taking the endpoints of the intervals over which p_i's$(1 \leq i \leq m)$ are executed. The construct permits the processes $p_1, \ldots, p_m, q$ to be autonomous. The existential quantification statement $\exists x : p(x)$ intends to hide the variable x within the process p. We use a renaming method to reduce this kind of programs. Consider a formula $\exists x : p(x)$ with a bound name x. The existential quantification $(\exists x)$ is removed from $\exists x : p(x)$ to obtain an equivalent formula $p(y)$ with a free variable y by renaming x as y, i.e., $\exists x : p(x) \stackrel{y}{\equiv} p(y)$.

In the following, we briefly introduce the operational semantics of MSVL programs (Yang and Duan 2008). The reduction process of MSVL programs is divided into two phases: one for state reduction and the other for interval reduction.

For obtaining the values of variables in a state, we use $s_i[(m, \dot{p_x})/x]$ to map x to a pair $(m, \dot{p_x})$ with other variables being not changed at state s_i. Here, $\dot{p_x}$ denotes assignment flag p_x or $\neg p_x$ and m is a value in D. Further, for convenience, we use $s_i[w]$ to mean that all variables and propositions appearing in state program w at current state s_i are instantiated with their values. There are two types of configurations, one for expressions and the other for programs. A configuration is defined in terms of a quadruple $\mathbb{C} = (p, \sigma_{i-1}, s_i, i)$, where p is a framed MSVL program, $\sigma_{i-1} = \langle s_0, \ldots, s_{i-1}\rangle (i > 1)$ a model which records information of all states, s_i the current state at which p is being executed and i a counter for counting the number of states in σ_{i-1}. Further, for i = 0, let $\sigma_{-1} = \epsilon$ be an empty sequence. Thus, the initial configuration is $\mathbb{C}_0 = (p, \epsilon, s_0, 0)$. When a program is terminable and satisfiable, its final configuration arrives at $\mathbb{C}_f = (\mathsf{true}, \sigma, \varnothing, |\sigma| + 1)$. Similarly, the configurations for arithmetic expression e and boolean expression b are $(e, \sigma_{i-1}, s_i, i)$ and $(b, \sigma_{i-1}, s_i, i)$ respectively. There are two types of relations over the set of configurations, one for configurations w.r.t. a state, and the other for configurations with different states. For configurations within a state, the notation $\rightarrowtail$ is used, whereas for configurations with different states, $\rightarrow$ is used.

Let CS be the set of all configurations. The other transition relations according to $\rightarrowtail$ and $\rightarrow$ are given as follows.

$$\stackrel{0}{\rightarrowtail} : \{(\mathbb{C}, \mathbb{C}) | \mathbb{C} \in CS\}$$

$$\stackrel{i+1}{\rightarrowtail} : \{(\mathbb{C}_1, \mathbb{C}_2) | \mathbb{C}_1, \mathbb{C}_2 \in CS \text{ and } \exists \mathbb{C}' \in CS, \mathbb{C}_1 \stackrel{i}{\rightarrowtail} \mathbb{C}' \text{ and } \mathbb{C}' \rightarrowtail \mathbb{C}_2, i \geq 0\}$$

$$\stackrel{+}{\rightarrowtail} : \bigcup_{i>0} \stackrel{i}{\rightarrowtail}$$

$$\stackrel{*}{\rightarrowtail} : \stackrel{0}{\rightarrowtail} \cup \stackrel{+}{\rightarrowtail}$$

$$\stackrel{0}{\rightarrow} : \{(\mathbb{C}, \mathbb{C}') | \mathbb{C}, \mathbb{C}' \in CS \text{ and } \mathbb{C} \stackrel{*}{\rightarrowtail} \mathbb{C}'\}$$

$$\stackrel{i+1}{\rightarrow} : \{(\mathbb{C}_1, \mathbb{C}_2) | \mathbb{C}_1, \mathbb{C}_2 \in CS \text{ and } \exists \mathbb{C}' \in CS, \mathbb{C}_1 \stackrel{i}{\rightarrow} \mathbb{C}' \text{ and } \mathbb{C}' \rightarrow \mathbb{C}_2, i \geq 0\}$$

$$\stackrel{+}{\rightarrow} : \bigcup_{i>0} \stackrel{i}{\rightarrow}$$

$$\stackrel{*}{\rightarrow} : \stackrel{0}{\rightarrow} \cup \stackrel{+}{\rightarrow}$$

Usually, we use CS_q to denote the set of the configurations produced from the MSVL program q. The evaluation rules of expressions, transition rules within a state and interval transition rules is defined in (Yang and Duan 2008).

There are two significant kinds of communication modes between concurrent processes: by shared variables and by channels. In MSVL, such a communication

is based on shared variables, whereas, in π-calculus, is by channels. For smoothness of transformation, the channel and communication primitives are defined in MSVL.To model mobility, a channel should be capable of being passed as a message. To this end, the channel is formalized as a pointer referring to a tripe. The channel and communication primitives are specified in Definition 3.

Definition 3

$$
\begin{array}{ll}
C_x & \stackrel{\text{def}}{=} \&X \\
X & \stackrel{\text{def}}{=} (w, r, v_p) \\
write(C_x) & \stackrel{\text{def}}{=} \Pi_1(*C_x) = \mathsf{true} \\
read(C_x) & \stackrel{\text{def}}{=} \Pi_2(*C_x) = \mathsf{true} \\
send(C_x, C_y) & \stackrel{\text{def}}{=} write(C_x) \wedge \Pi_3(*C_x) = C_y \wedge \varepsilon \\
receive(C_x, C_y) & \stackrel{\text{def}}{=} read(C_x) \wedge C_y = \Pi_3(*C_x) \wedge \varepsilon \\
wait\_send(C_x, C_y) & \stackrel{\text{def}}{=} write(C_x) \wedge await(read(C_x)); (\Pi_3(*C_x) = C_y \wedge \varepsilon) \\
wait\_receive(C_x, C_y) & \stackrel{\text{def}}{=} read(C_x) \wedge await(write(C_x)); (C_y = \Pi_3(*C_x) \wedge \varepsilon)
\end{array}
$$

where w and r are boolean variables declaring weather or not channel C_x is waiting for sending a message or receiving a message. v_p is a channel variable representing the message to be sent by C_x . Here, X is a tripe connected with channel C_x and cannot be used for other purposes. Projection function $\Pi_i (1 \leq i \leq 3)$ is defined to obtain a component of the multi-components, e.g., $\Pi_1(X) = \Pi_1(w, r, v_p) = w$. $write(C_x)$ denotes a send action occurring via channel C_x and $read(C_x)$ a receive action occurring via channel C_x. Communications can be synchronous or asynchronous. $send(C_x, C_y)$ represents a asynchronous sending primitive, $receive(C_x, C_y)$ a asynchronous receiving primitive, $wait\_send(C_x, C_y)$ a synchronous sending primitive and $wait\_receive(C_x, C_y)$ a synchronous receiving primitive. Asynchronous sending and receiving will execute writing or reading instantaneously while synchronous sending and receiving will wait for some actions from its partner.

4 Transformation from p-π to MSVL

Although p-π and MSVL describe concurrent reactive systems fundamentally in different ways, we can show that each bounded p-π process can be transformed into an MSVL program by the mapping function $\mathcal{F} : \{bounded\ p\text{-}\pi\ process\} \rightarrow \{\text{MSVL}program\}$, which is defined as follows. $\mathcal{F}(P)$ transforms P by the induction on its structures. The definition of bounded p-π process is given below.

Definition 4 Let P be a p-π process. P is bounded iff for each process identifier $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A$ with recursive calls in P, there is a process P'_A in the form of $\pi_1^s + \cdots + \pi_i^s + \pi_1^{'s} \cdot A\langle x_1, \ldots, x_n\rangle + \cdots + \pi_m^{'s} \cdot A\langle x_1, \ldots, x_n\rangle$ such that $A(x_1, \ldots, x_n) \equiv P'_A$ where, π_j^s $(1 \leq j \leq i)$ and $\pi_k^{'s}$ $(1 \leq k \leq m)$ denote $\pi_{j_1} \cdot \pi_{j_2} \cdot \cdots \cdot \pi_{j_{nj}}$ and $\pi_{k_1} \cdot \pi_{k_2} \cdot \ldots \cdot \pi_{k_{nk}}$, respectively.

For easy of discussion, we first consider names and propositions. According to the set of names $n(P)$ and propositions PR_P appearing in a p-π process P, the set of channels $CH_{\mathcal{F}(P)}$ and propositional (or boolean) variables $V_{\mathcal{F}(P)}$ of the corresponding MSVL program are obtained. For instance, if $n(P)$={a, b, c} and PR_P={p_1, p_2, p_3}, we have, $CH_{\mathcal{F}(P)} = \{C_a, C_b, C_c\}$ and $V_{\mathcal{F}(P)} = \{x_{p_1}, x_{p_2}, x_{p_3}\}$. The structural transformation rules of processes are formalized below.

4.1 Empty process

$$\mathcal{F}(0) \stackrel{\text{def}}{=} \varepsilon$$

Empty process 0 means doing nothing, so it can naturally be mapped to ε in MSVL.

4.2 Output action prefix

$$\mathcal{F}(\bar{x}\langle y\rangle \cdot P) \stackrel{\text{def}}{=} \mathit{send}(C_x, C_y); \mathcal{F}(P)$$

Output action prefix guarded process $\bar{x}\langle y\rangle \cdot P$ means it sends y through x and then behaves as P. Since $\bar{x}\langle y\rangle$ is an instantaneous action prefix without waiting for its partner, the process can be mapped to $\mathit{send}(C_x, C_y); \mathcal{F}(P)$ where $\mathit{send}(C_x, C_y)$ is an asynchronous sending. C_x and C_y are the corresponding MSVL channels of x and y.

4.3 Input action prefix

$$\mathcal{F}(x(y) \cdot P) \stackrel{\text{def}}{=} \exists C_y : (\mathsf{frame}(C_y) \wedge (\mathit{receive}(C_x, C_y); \mathcal{F}(P)))$$

Input action prefix guarded process $x(y) \cdot P$ indicates it will accept z from x and then behave as $\{z/y\}P$. Here, since y is a bound name and will be substituted later, it should be translated to a local channel. Therefore, the exist quantifier $\exists$ is brought in. Further, $\mathsf{frame}(C_y)$ guarantees that the channel C_y always keeps its old value (or refers to the old channel instance) over an interval if no assignment to C_y is encountered. What accurately happens is that C_y will be assigned the message sent by channel C_x. We assume the message is C_z so that C_y will refer to the same channel instance as C_z. Analogous to $\bar{x}\langle y\rangle$, $x(y)$ is translated into an asynchronous receiving.

4.4 Await

$$\mathcal{F}(\mathit{await}(\bar{x}\langle y\rangle \cdot P)) \stackrel{\text{def}}{=} \mathit{wait\_send}(C_x, C_y); \mathcal{F}(P)$$

$$\mathcal{F}(\mathit{await}(x(y) \cdot P)) \stackrel{\text{def}}{=} \exists C_y : (\mathsf{frame}(C_y) \wedge (\mathit{wait\_receive}(C_x, C_y); \mathcal{F}(P)))$$

For derived process $\mathit{await}(P)$ where P is a sending (or receiving) prefix guarded process, since the communication is synchronous, its transformation involves synchronous sending and receiving. Except for the different communication modes, the

transformations of $await\ (\bar{x}\langle y\rangle \cdot P)$ and $await(x(y) \cdot P)$ are similar to those of $\bar{x}\langle y\rangle \cdot P$ and $x(y) \cdot P$, respectively.

4.5 Internal action prefix

$$\mathcal{F}(\tau \cdot P) \stackrel{\text{def}}{=} p_I \wedge \varepsilon; \mathcal{F}(P)$$

Internal action prefix guarded process $\tau \cdot P$ performs an internal action τ and then behaves as P. It can be mapped to $p_I \wedge \varepsilon; \mathcal{F}(P)$ in which p_I is a special proposition used to manifest whether or not an internal action takes place.

4.6 Property action prefix

$$\mathcal{F}(I_p \cdot P) \stackrel{\text{def}}{=} x_{p_1} = \mathsf{true}\ \wedge \cdots \wedge x_{p_n} = \mathsf{true} \wedge skip; \mathcal{F}(P)$$

where $I_p = \{\tilde{p}_1, \ldots, \tilde{p}_n\}$. The property action prefix guarded process $I_p \cdot P$ satisfies $p_i (1 \leq i \leq n)$ in the first time unit and then behaves as P. Intuitively, it can be mapped to x_{p_1}=true $\wedge \cdots \wedge x_{p_n}$=true $\wedge\ skip; \mathcal{F}(P)$. Here, $x_{p_1}, \ldots,$ and x_{p_n} are the corresponding boolean variables of $p_1, \ldots,$ and p_n, respectively.

4.7 Time unit action prefix

$$\mathcal{F}(skip \cdot P) \stackrel{\text{def}}{=} skip; \mathcal{F}(P)$$

Time unit action prefix guarded process $skip \cdot P$ declares that it will idle in the first time unit and then behave as P. Its transformation is straightforward.

4.8 Empty action prefix

$$\mathcal{F}(\varepsilon \cdot P) \stackrel{\text{def}}{=} \varepsilon; \mathcal{F}(P)$$

For empty action prefix guarded process $\varepsilon \cdot P$, it will execute an empty transition and then behave as P. Similar to $skip \cdot P$, it is directly translated into $\varepsilon; \mathcal{F}(P)$.

4.9 Nondeterministic choice

$$\mathcal{F}(P_1 + P_2) \stackrel{\text{def}}{=} \mathcal{F}(P_1) \vee \mathcal{F}(P_2)$$

Structure $P_1 + P_2$ shows that P_1 and P_2 will proceed nondeterminately, so that it can naturally be expressed by a disjunction statement.

4.10 Parallel

$$\mathcal{F}(P_1 \mid P_2) \stackrel{\text{def}}{=} \mathcal{F}(P_1) \,||\, \mathcal{F}(P_2)$$

Parallel composition structure $P_1 \mid P_2$ manifests that P_1 and P_2 execute in parallel and will directly be transformed into a parallel statement.

4.11 Match

$$\mathcal{F}([a_1 = a_2]P) \stackrel{\text{def}}{=} \texttt{if}\ C_{a_1} = C_{a_2}\ \texttt{then}\ \mathcal{F}(P)\ \texttt{else}\ \varepsilon$$

Match structure $[a_1 = a_2]P$ tells us that if a_1 and a_2 are the same name, the process will behave as P, or else it will as 0. Its transformation is defined based on the conditional statement.

4.12 Restriction

$$\mathcal{F}(\nu a\ P) \stackrel{\text{def}}{=} \exists C_a : (\mathsf{frame}(C_a) \wedge \mathcal{F}(P))$$

Restriction structure $\nu a\ P$ means that the process will behave as P except that the communication on bound name a is forbidden. a will be mapped to local channel C_a so that restriction structure $\nu a\ P$ can be defined above, where frame is used as in the transformation of the input action prefix guarded structure.

4.13 Process instance

For process instance $A\langle a_1, \ldots, a_n\rangle$, we assume that $A(x_1, \ldots, x_n)$ is defined by P_A, so that $A\langle a_1, \ldots, a_n\rangle = \{\overrightarrow{a}/\overrightarrow{x}\}P_A$. To transform process instance, we firstly claim a theorem below.

Theorem 1 *Let process* $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P_A$ *and* $A\langle a_1, \ldots, a_n\rangle = \{\overrightarrow{a}/\overrightarrow{x}\}P_A$. *If* A *is recursively called in* P_A *and there is a* P'_A *in the form of* $\pi_1^s + \cdots + \pi_i^s + \pi_1^{'s} \cdot A\langle x_1, \ldots, x_n\rangle + \cdots . + \pi_m^{'s} \cdot A\langle x_1, \ldots, x_n\rangle$ *such that* $A(x_1, \ldots, x_n) \equiv P'_A$, *where,* π_j^s $(1 \le j \le i)$ *and* $\pi_k^{'s}$ $(1 \le k \le m)$ *denote* $\pi_{j_1} \cdot \cdots \cdot \pi_{j_{nj}}$, $\pi_{k_1} \cdot \cdots \cdot \pi_{k_{nk}}$, *respectively, then, there exists the least fixed point* $L_{fix}\langle a_1, \ldots, a_n\rangle$ *of* $A\langle a_1, \ldots, a_n\rangle$ *such that* $A\langle a_1, \ldots, a_n\rangle = L_{fix}\langle a_1, \ldots, a_n\rangle$.

Proof The theorem can be proved by using Tarski's fixed-point theorem (Tarski 1955) and Scott's fixed-point induction (Winskel 1993). □

From above, for the simple definition $A(x_1, \ldots, x_n) = \pi_1^s + \pi_1^{'s} \cdot A\ \langle x_1, \ldots, x_n\rangle$, the least fixed point of $A\langle a_1, \ldots, a_n\rangle$ is $\{\overrightarrow{a}/\overrightarrow{x}\}\pi_1^{'s*} \cdot \pi_1^s$. However, in practice, the iterative number of times is deterministic. Therefore, to transform $A\langle a_1, \ldots, a_n\rangle$ into

MSVL, we can specify an arbitrary integer N as number of iteration times to control the recursion. For $A(x_1,\ldots,x_n)=\pi_1^s+\cdots+\pi_i^s+\pi_1^{'s}\cdot A\langle x_1,\ldots,x_n\rangle+\cdots.+\pi_m^{'s}\cdot A\langle x_1,\ldots,x_n\rangle$, the transformation is similar. Since a process instance can be defined with recursive calls in p-π, its transformation will be divided into three cases:

1. Without recursive calls in the definition of A:

$$\mathcal{F}(A\langle a_1,\ldots,a_n\rangle)\stackrel{\text{def}}{=}\{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(P_A)$$

   For this kind of process instance, it will be directly translated by substitution. $\overrightarrow{C_a}$ and $\overrightarrow{C_x}$ are the corresponding MSVL channel vectors of $\overrightarrow{a}$ and $\overrightarrow{x}$. Here, we use a substituting method $\{\overrightarrow{C_a}/\overrightarrow{C_x}\}$ to substitute $\overrightarrow{C_a}$ for $\overrightarrow{C_x}$ so that each C_{x_i} will be substituted by C_{a_i} within the bound scope of $C_{x_i}(1\le i\le n)$ in $\mathcal{F}(P_A)$.
2. With recursive calls in A and $A(x_1,\ldots,x_n)\equiv\pi_1^s+\pi_1^{'s}\cdot A\langle x_1,\ldots,x_n\rangle$:

$$\begin{aligned}\mathcal{F}(A\langle a_1,\ldots,a_n\rangle)\stackrel{\text{def}}{=}\ & j:=0;\\ & \mathcal{F}(A_{-1}\langle a_1,\ldots,a_n\rangle):=\varepsilon;\\ & \mathcal{F}(A_0\langle a_1,\ldots,a_n\rangle):=\{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^s);\\ & \texttt{while}\ (j<N)\\ & \texttt{do}\ (j:=j+1;\\ & \mathcal{F}(A_j\langle a_1,\ldots,a_n\rangle):=\{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^s)\\ & \vee(\{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^{'s});\mathcal{F}(A_{j-1}\langle a_1,\ldots,a_n\rangle)))\end{aligned}$$

   Where N is a constant integer. For this kind process instance, by Theorem 1, a least fixed point can be obtained. In practice, we employ a recursion procedure to get an arbitrary given times iteration result of $A\langle a_1,\ldots,a_n\rangle$. For process instances with recursive calls in the definition of A and $A(x_1,\ldots,x_n)=\pi_1^s+\cdots+\pi_i^s+\pi_1^{'s}\cdot A\langle x_1,\ldots,x_n\rangle+\cdots.+\pi_m^{'s}\cdot A\langle x_1,\ldots,x_n\rangle$, the corresponding MSVL programs can be obtained in the same way.
3. With recursive calls in the definition of A but A is not in the form stated in Theorem 1: For this kind process identifier, its behavior may not be convergent so that we cannot directly transform them into MSVL.

According to the transformation $\mathcal{F}$, we investigate the consistency between interleaving and true concurrency. In p-π the execution model of instantaneous action prefixes in parallel structure is in interleaving mode whereas in MSVL the parallel statement $f_1||f_2$ is reduced in true concurrency mode. By the following proofs, it is obtained that the two kinds of concurrency mechanisms are consistent in terms of the transformation. The execution of p-π processes comprises two stages: one for instantaneous action prefixes and the other for interval action prefixes. Coincidentally, the reduction of MSVL programs is divided into two phases: one for state reduction and the other for interval reduction. We will show the following facts: (1) the execution of instantaneous action prefixes is consistent with the state reduction; (2) the execution of interval action prefixes is unified with the interval reduction.

(1) For the execution of instantaneous action prefixes, there are three possible conditions: (a) communicating free names; (b) communicating bound names; (c) internal communication. The proof with respect to the three conditions is given as follows.

(a) communicating free names Let $P_1 = \bar{x}\langle y\rangle \cdot P_1'$ and $P_2 = x(z) \cdot P_2'$, we have,

$$\frac{\dfrac{}{\bar{x}\langle y\rangle \cdot P_1' \xrightarrow{\bar{x}\langle y\rangle} P_1'}(\text{by Out}) \quad \dfrac{}{x(z) \cdot P_2' \xrightarrow{x(w)} \{w/z\}P_2'}(\text{by In})}{P_1|P_2 \xrightarrow{\tau} P_1'|\{y/w\}\{w/z\}P_2'}(\text{by Com}).$$

For $\mathcal{F}(P_1|P_2)$, we have,

$$\begin{aligned}
\mathcal{F}(P_1|P_2) &\equiv \mathcal{F}(\bar{x}\langle y\rangle \cdot P_1'|x(z) \cdot P_2') \\
&\equiv \mathcal{F}(\bar{x}\langle y\rangle \cdot P_1')||\mathcal{F}(x(z) \cdot P_2') \\
&\equiv (\mathit{send}(C_x, C_y); \mathcal{F}(P_1'))||\exists C_z : (\mathsf{frame}(C_z) \wedge (\mathit{receive}(C_x, C_z); \mathcal{F}(P_2'))) \\
&\overset{C_w}{\equiv} (\mathit{send}(C_x, C_y); \mathcal{F}(P_1'))||\mathsf{frame}(C_w) \wedge (\mathit{receive}(C_x, C_w); \{C_w/C_z\}\mathcal{F}(P_2')) \\
&\equiv \mathsf{frame}(C_w) \wedge ((\mathit{send}(C_x, C_y); \mathcal{F}(P_1'))||(\mathit{receive}(C_x, C_w); \{C_w/C_z\}\mathcal{F}(P_2'))) \\
&\equiv^* \mathsf{frame}(C_w) \wedge (\Pi_1(*C_x) = \mathsf{true} \wedge \Pi_2(*C_x) = \mathsf{true} \wedge C_w = C_y \wedge \varepsilon \wedge (\mathcal{F}(P_1') \\
&\quad ||\{C_w/C_z\}\mathcal{F}(P_2'))).
\end{aligned}$$

Focus on the obtained formula, since $\Pi_1(*C_x) = \mathsf{true} \wedge \Pi_2(*C_x) = \mathsf{true} \wedge C_w = C_y \wedge \varepsilon$ is a state formula, i.e., without any temporal operators in the formula, communicating free names is consistent with the state reduction.

(b) communicating bound names Let $P_1 = \nu y\ \bar{x}\langle y\rangle \cdot P_1'$ and $P_2 = x(z) \cdot P_2'$, it is obtained,

$$\frac{\dfrac{\dfrac{}{\bar{x}\langle y\rangle \cdot P_1' \xrightarrow{\bar{x}\langle y\rangle} P_1'}(\text{by Out})}{\nu y\ \bar{x}\langle y\rangle \cdot P_1' \xrightarrow{\bar{x}(w)} \{w/y\}P_1'}(\text{by Open}) \quad \dfrac{}{x(z) \cdot P_2' \xrightarrow{x(u)} \{u/z\}P_2'}(\text{by In})}{\nu y\ \bar{x}\langle y\rangle \cdot P_1'|x(z) \cdot P_2' \xrightarrow{\tau} \nu w\ (\{w/y\}P_1'|\{w/u\}\{u/z\}P_2')}(\text{by Close}).$$

For $\mathcal{F}(P_1|P_2)$, we have,

$$\begin{aligned}
\mathcal{F}(P_1|P_2) &\equiv \mathcal{F}(\nu y\ \bar{x}\langle y\rangle \cdot P_1'|x(z) \cdot P_2') \\
&\equiv \mathcal{F}(\nu y\ \bar{x}\langle y\rangle \cdot P_1')||\mathcal{F}(x(z) \cdot P_2') \\
&\equiv \exists C_y : (\mathsf{frame}(C_y) \wedge \mathcal{F}(\bar{x}\langle y\rangle \cdot P_1'))||\mathcal{F}(x(z) \cdot P_2') \\
&\equiv \exists C_y : (\mathsf{frame}(C_y) \wedge (\mathit{send}(C_x, C_y); \mathcal{F}(P_1')))||\exists C_z : (\mathsf{frame}(C_z) \wedge \\
&\quad (\mathit{receive}(C_x, C_z); \mathcal{F}(P_2'))) \\
&\overset{C_w}{\equiv} (\mathsf{frame}(C_w) \wedge (\mathit{send}(C_x, C_w); \{C_w/C_y\}\mathcal{F}(P_1')))||\exists C_z : (\mathsf{frame}(C_z) \\
&\wedge\ (\mathit{receive}(C_x, C_z); \mathcal{F}(P_2')))
\end{aligned}$$

$$
\begin{aligned}
&\overset{C_u}{\equiv} \mathsf{frame}(C_w) \wedge (\mathit{send}(C_x, C_w); \{C_w/C_y\}\mathcal{F}(P_1'))||\mathsf{frame}(C_u) \wedge \\
&\quad (\mathit{receive}(C_x, C_u); \{C_u/C_z\}\mathcal{F}(P_2')) \\
&\equiv \mathsf{frame}(C_w) \wedge \mathsf{frame}(C_u) \wedge (\mathit{send}(C_x, C_w); \{C_w/C_y\}\mathcal{F}(P_1'))|| \\
&\quad (\mathit{receive}(C_x, C_u); \{C_u/C_z\}\mathcal{F}(P_2')) \\
&\equiv^* \mathsf{frame}(C_w) \wedge \mathsf{frame}(C_u) \wedge (\Pi_1(*C_x) = \mathsf{true} \wedge \Pi_2(*C_x) = \mathsf{true} \\
&\wedge\ C_u = C_w \wedge \varepsilon \wedge (\{C_w/C_y\}\mathcal{F}(P_1')||\{C_u/C_z\}\mathcal{F}(P_2'))
\end{aligned}
$$

Similarly, since $\Pi_1(*C_x) = \mathsf{true} \wedge \Pi_2(*C_x) = \mathsf{true} \wedge C_u = C_w \wedge \varepsilon$ is a state formula, communicating bound names is consistent with the state reduction.

(c) internal communication For $\tau \cdot P$, we have,

$$\frac{}{\tau \cdot P \overset{\tau}{\rightarrow} P}(\text{by Tau}).$$

For $\mathcal{F}(\tau \cdot P)$, it is obtained, $\mathcal{F}(\tau \cdot P) \equiv p_I \wedge \varepsilon; \mathcal{F}(P)$.

Obviously, $p_I \wedge \varepsilon$ is a state formula. Therefore, internal communication is consistent with the state reduction.

(2) For the execution of interval action prefixes, let $P_1 = \pi_{t_1} \cdot P_1'$ and $P_2 = \pi_{t_2} \cdot P_2'$, we have,

$$\frac{\dfrac{}{P_1 \overset{\pi_{t_1}}{\longrightarrow} P_1'}(\text{by Act}_t), \quad \dfrac{}{P_2 \overset{\pi_{t_2}}{\longrightarrow} P_2'}(\text{by Act}_t)}{P_1|P_2 \overset{\pi_{t_1} \cup \pi_{t_2}}{\longrightarrow} P_1'|P_2'}(\text{by Com}_t).$$

We assume $\pi_{t_1} = \{\tilde{p}_1, \tilde{p}_2\}$ and $\pi_{t_2} = \{\tilde{p}_3, \tilde{p}_4\}$. For $\mathcal{F}(P_1|P_2)$, we have,

$$
\begin{aligned}
\mathcal{F}(P_1|P_2) &\equiv \mathcal{F}(\{\tilde{p}_1, \tilde{p}_2\} \cdot P_1'|\{\tilde{p}_3, \tilde{p}_4\} \cdot P_2') \\
&\equiv \mathcal{F}(\{\tilde{p}_1, \tilde{p}_2\} \cdot P_1')||\mathcal{F}(\{\tilde{p}_3, \tilde{p}_4\} \cdot P_2') \\
&\equiv (x_{p_1}{=}\mathsf{true} \wedge x_{p_2}{=}\mathsf{true} \wedge \mathit{skip}; \mathcal{F}(P_1'))||(x_{p_3}{=}\mathsf{true} \wedge x_{p_4}{=}\mathsf{true} \wedge \mathit{skip}; \\
&\quad \mathcal{F}(P_2')) \\
&\equiv^* x_{p_1}{=}\mathsf{true} \wedge x_{p_2}{=}\mathsf{true} \wedge x_{p_3}{=}\mathsf{true} \wedge x_{p_4}{=}\mathsf{true} \wedge \bigcirc(\mathcal{F}(P_1')||\mathcal{F}(P_2'))
\end{aligned}
$$

By the transformation $\mathcal{F}$ and the corresponding semantic equivalence rules of MSVL, the obtained formula is of the form $q_c \wedge \bigcirc q_f$. This means the reduction of the formula need to move to the next state and interval transition rules are required to work with the reduction process. The execution of processes with skip is similar. Thus, the execution of interval action prefixes is consistent with the interval reduction.

Here, we discuss two notices related to the transformation.

(1) The variation of the priority levels.

In MSVL, the priority levels of operators are defined as follows with 1=highest and 7=lowest:

Table 1 Correspondence between operators

In p-π	In MSVL
$.$	$;$
ν	$\exists$
$+$	$\vee$
$\mid$	$\mid\mid$

Table 2 Correspondence between algebraic laws

In p-π	In MSVL
$\nu x\ P \equiv \nu y\ \{y/x\}P$	$\exists x : f(x) \equiv \exists y : f(y)$
$P + Q \equiv Q + P$	$f_1 \vee f_2 \equiv f_2 \vee f_1$
$P \mid Q \equiv Q \mid P$	$P \mid\mid Q \equiv Q \mid\mid P$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$(P \mid\mid Q) \mid\mid R \equiv P \mid\mid (Q \mid\mid R)$
$\nu x\ 0 \equiv 0$	$\exists x : \varepsilon \equiv \varepsilon$
$\nu x \nu y\ P \equiv \nu y \nu x\ P$	$\exists x : \exists y : f(x, y) \equiv \exists y : \exists x : f(x, y)$

$1: \neg \quad 2: \bigcirc, \Box, \Diamond, \odot, +, * \quad 3: \exists, \forall \quad 4: =$
$5: \vee, \wedge \quad 6: \rightarrow, \leftrightarrow \quad 7: ;, \mathsf{prj}$

In p-π, prefixed operations ($\pi\cdot$, νa and $[a_1 = a_2]$) have priority over summation $+$ and composition $|$.

By the transformation $\mathcal{F}$, these is a correspondence between operators of p-π and MSVL in Table 1.

Note that, in p-π, the precedence of "$\cdot$" is higher than "$+$". However, in MSVL, the precedence of "$;$" is lower than "$\vee$". Therefore, several parentheses will be required during the transformation to avoid unconsistency and ambiguity.

(2) The correspondence between the algebraic laws of p-π and MSVL.

Table 2 shows the correspondence between some algebraic laws of p-π and MSVL.

5 Soundness of transformation

To prove the soundness of the transformation, we first give the definition of bisimulation over the LTS of a p-π process and the set of reduction configurations of an MSVL program. Based on the definition, the soundness of the transformation is proven.

Definition 5 A binary relation B' over the processes of LTS_P and the elements of the set of MSVL reduction configurations $\mathrm{CS}_{\mathcal{F}(P)}$ is a bisimulation iff $(P, \mathbb{C}_0) \in B'$, where $\mathbb{C}_0$ is the initial configuration of $\mathcal{F}(P)$, and if $(P', \mathbb{C}') \in B'$ then the following hold:

- if $P' \xrightarrow{\pi_c} P''$ then there exists $\mathbb{C}''$ such that $\mathbb{C}' \overset{*}{\rightarrowtail} \mathbb{C}''$ where $\mathbb{C}' = (p', \sigma', s', i')$ and $\mathbb{C}''=(p'', \sigma'', s'', i'')$ with $p' \equiv \mathcal{F}(\pi_c)$; p'', $\sigma''=\sigma'$, $s''=s'[w_{\mathcal{F}(\pi_c)}]$ ($w_{\mathcal{F}(\pi_c)}$ is the corresponding state program of $\mathcal{F}(\pi_c)$) and $i'' = i'$, and $(P'', \mathbb{C}'') \in B'$;
- if $P' \xrightarrow{\pi_t} P''$ then there exists $\mathbb{C}''$ such that $\mathbb{C}' \rightarrow \mathbb{C}''$, where $\mathbb{C}' = (p', \sigma', s_{i'}, i')$ and $\mathbb{C}'' = (p'', \sigma'', s_{i'+1}, i' + 1)$ with $p' \equiv \mathcal{F}(\pi_t)$; p'' and $\sigma'' = \sigma'.\langle s_{i'}[w_{\mathcal{F}(\pi_t)}]\rangle$ ($w_{\mathcal{F}(\pi_t)}$ is the corresponding state program of $\mathcal{F}(\pi_t)$), and $(P'', \mathbb{C}'') \in B'$;

– if $\mathbb{C}' \overset{*}{\rightarrowtail} \mathbb{C}''$ where $\mathbb{C}' = (p', \sigma', s', i')$ and $\mathbb{C}'' = (p'', \sigma'', s'', i'')$ with $p' \equiv p_c; p''$, $\sigma'' = \sigma', s'' = s'[w_{p_c}]$ (p_c represents a communication primitive $send(C_x, C_y)$ or $receive(C_x, C_y)$) and $i'' = i'$, then there exists P'' such that $P' \overset{\pi_{p_c}}{\Longrightarrow} P''$ where π_{p_c} is the corresponding p-π instantaneous action prefix of p_c and $(\mathbb{C}'', P'') \in B'$;
– if $\mathbb{C}' \rightarrow \mathbb{C}''$ where $\mathbb{C}' = (p', \sigma', s_{i'}, i')$ and $\mathbb{C}'' = (p'', \sigma'', s_{i'+1}, i' + 1)$ with $p' \equiv x_{p_1}$=true $\wedge \cdots \wedge x_{p_n}$=true $\wedge \bigcirc p''$ and σ''=$\sigma'.\langle s_{i'}[(\mathsf{true}, \neg p_{x_{p_1}})/x_{p_1}] \cdots [(\mathsf{true}, \neg p_{x_{p_n}})/x_{p_n}]\rangle$) or $p' \equiv \bigcirc p''$ and σ''=$\sigma'.\langle s_{i'}\rangle$, then there exists P'' such that $P' \overset{I_p}{\Longrightarrow} P''$ where I_p=$\{\tilde{p}_1, \ldots, \tilde{p}_n\}$ or $P' \overset{skip}{\Longrightarrow} P''$ and $(\mathbb{C}'', P'') \in B'$.

where the relation $\overset{\pi'}{\Rightarrow}$ for any $\pi' \in \{\overline{x}\langle y\rangle, x(y), \overline{x}(y), I_p, skip\}$ is defined as follows:

1. $P \Rightarrow Q$ means that there is a sequence of zero or more internal actions $P \overset{\tau}{\rightarrow} P_1 \cdots P_n \overset{\tau}{\rightarrow} Q$. Formally, $\Rightarrow \overset{\text{def}}{=} \overset{\tau}{\rightarrow}^*$, the transitive reflexive closure of $\overset{\tau}{\rightarrow}$.
2. $P \overset{\pi'}{\Rightarrow} Q$ means $P \Rightarrow P_1 \overset{\pi'}{\rightarrow} P_n \Rightarrow Q$. Formally, $\overset{\pi'}{\Rightarrow} \overset{\text{def}}{=} \Rightarrow \overset{\pi'}{\rightarrow} \Rightarrow$.

If $(P', \mathbb{C}') \in B'$, we say that $\mathbb{C}'$ simulates P'. Similarly, if $(\mathbb{C}', P') \in B'$, we say that P' simulates $\mathbb{C}'$.

Theorem 2 *The transformation is soundness. More precisely, for each p-π process P, there is a bisimulation between the processes of LTS_P generated by the execution of P and configurations of $CS_{\mathcal{F}(P)}$ produced by MSVL program $\mathcal{F}(P)$ reductions.*

Proof The proof proceeds by induction on the structure of p-π processes. As you can see, the transformation from process expressions to MSVL programs ensures that for every transition in the former, one can find corresponding transitions in the latter and vice versa. It is then a matter of case by case analysis to conclude the two corresponding specifications simulate each other closely. Since the proofs of P simulates $\mathbb{C}_0$ and $\mathbb{C}_0$ simulates P are similar, we only prove $\mathbb{C}_0$ simulates P here.

Base (1) For empty process 0: since it does nothing, it can be simulated by the configuration $\mathbb{C} = (\varepsilon, \epsilon, s_0, 0)$.

(2) For instantaneous action prefix guarded structure $\bar{x}\langle y\rangle \cdot P$: the unique transition it can perform is $\bar{x}\langle y\rangle \cdot P \overset{\bar{x}\langle y\rangle}{\longrightarrow} P$ and $\mathcal{F}(\bar{x}\langle y\rangle \cdot P) \overset{\text{def}}{=} send(C_x, C_y); \mathcal{F}(P)$. For this transition of p-π, we can find the corresponding transitions of MSVL within a state $\mathbb{C} \overset{*}{\rightarrowtail} \mathbb{C}'$ (in which $\mathbb{C}$=(p, σ, s, i), $\mathbb{C}'$=(p', σ', s', i') and $p \equiv \mathcal{F}(\bar{x}\langle y\rangle)$; p', σ'=σ, s'=$s[w_{\mathcal{F}(\bar{x}\langle y\rangle)}]$, i'=i). The proofs for $x(y) \cdot P$, $\tau \cdot P$ and $\varepsilon \cdot P$ are similar.

(3) For interval action prefix guarded structure $I_p \cdot P$: the transition it can execute is $I_p \cdot P \overset{I_p}{\rightarrow} P$ and $\mathcal{F}(I_p \cdot P) \overset{\text{def}}{=} x_{p_1}$=true $\wedge \cdots \wedge x_{p_n}$=true $\wedge\, skip; \mathcal{F}(P) \equiv x_{p_1}$=true $\wedge \cdots \wedge x_{p_n}$=true $\wedge \bigcirc \mathcal{F}(P)$. Similarly, for this p-$\pi$ transition, the obtained corresponding MSVL transitions are $\mathbb{C}' \rightarrow \mathbb{C}''$ where $\mathbb{C}'$=$(p', \sigma', s_{i'}, i')$ and $\mathbb{C}''$=$(p'', \sigma'', s_{i'+1}, i' + 1)$ with $p' \equiv \mathcal{F}(I_p)$; p'' and $\sigma'' = \sigma'.\langle s_{i'}[(\mathsf{true}, \neg p_{x_{p_1}})/x_{p_1}] \cdots [(\mathsf{true}, \neg p_{x_{p_n}})/x_{p_n}]\rangle$. For $skip \cdot P$, the only transition it can perform is $skip \cdot P \overset{skip}{\longrightarrow} P$ and $\mathcal{F}(skip \cdot P) \overset{\text{def}}{=} skip; \mathcal{F}(P) \equiv \bigcirc P$. For this p-$\pi$ transition, the found corresponding MSVL transition is $\mathbb{C}' \rightarrow \mathbb{C}''$ where $\mathbb{C}' = (p', \sigma', s_{i'}, i')$ and $\mathbb{C}'' = (p'', \sigma'', s_{i'+1}, i' + 1)$ with $p' \equiv \mathcal{F}(skip)$; p'' and $\sigma'' = \sigma'.\langle s_{i'}\rangle$.

Induction (1) For nondeterministic choice structure $P_1 + P_2$: transitions it can execute depend on P_1 and P_2. We assume that $P_1 \xrightarrow{\pi_{o_1}} P_1'$ and $P_2 \xrightarrow{\pi_{o_2}} P_2'$, and further they are simulated by transitions $\mathbb{C}_1 \xrightarrow{*} \mathbb{C}_1'$ and $\mathbb{C}_2 \xrightarrow{*} \mathbb{C}_2'$ respectively. Since $\mathcal{F}(P_1+P_2) \stackrel{\text{def}}{=} \mathcal{F}(P_1) \vee \mathcal{F}(P_2)$, what transitions P_1+P_2 can perform will be simulated by transitions $\mathbb{C} \xrightarrow{*} \mathbb{C}_1'$ and $\mathbb{C} \xrightarrow{*} \mathbb{C}_2'$ where $\Pi_1(\mathbb{C}) = \mathcal{F}(P_1 + P_2)$, $\Pi_1(\mathbb{C}_1) = \mathcal{F}(P_1)$ and $\Pi_1(\mathbb{C}_2) = \mathcal{F}(P_2)$. Here the projection function Π_1 is used as before.

(2) For parallel structure $P_1|P_2$: transitions it can execute similarly depend on P_1 and P_2. We assume that $P_1 \xrightarrow{\pi_{o_1}} P_1'$ and $P_2 \xrightarrow{\pi_{o_2}} P_2'$, and they are likewise simulated by transitions $\mathbb{C}_1 \xrightarrow{*} \mathbb{C}_1'$ and $\mathbb{C}_2 \xrightarrow{*} \mathbb{C}_2'$ respectively. $\mathcal{F}(P_1|P_2) \stackrel{\text{def}}{=} \mathcal{F}(P_1) \,||\, \mathcal{F}(P_2)$, so that we have that what transitions $P_1|P_2$ can execute will be simulated by transitions $\mathbb{C} \xrightarrow{*} \mathbb{C}_1' \Uparrow \mathbb{C}_2$, $\mathbb{C} \xrightarrow{*} \mathbb{C}_1 \Uparrow \mathbb{C}_2'$ or $\mathbb{C} \xrightarrow{*} \mathbb{C}_1' \Uparrow \mathbb{C}_2'$, where $\Pi_1(\mathbb{C}) = \mathcal{F}(P_1|P_2)$, $\Pi_1(\mathbb{C}_1) = \mathcal{F}(P_1)$ and $\Pi_1(\mathbb{C}_2) = \mathcal{F}(P_2)$. We use the operator $\Uparrow$ to conjoin two configurations. For instance, to conjoin configurations $\mathbb{C} = (p, \sigma, s, i)$ and $\mathbb{C}' = (p', \sigma', s', i')$ (Here σ'=σ, s'=s and i'=i and they are the conditions must be satisfied by two conjoint configurations), we have $\mathbb{C} \Uparrow \mathbb{C}' \stackrel{\text{def}}{=} (p||p', \sigma, s, i)$.

(3) For match structure $[a_1 = a_2]P$: transitions it can execute are dependent on P. If the names a_1 and a_2 are equivalent, the behavior of $[a_1 = a_2]P$ should be as P, or else as 0. We assume that $P \xrightarrow{\pi_o} P'$ and this transition will be simulated by transitions $\mathbb{C}' \xrightarrow{*} \mathbb{C}''$. According to $\mathcal{F}([a_1 = a_2]P) \stackrel{\text{def}}{=}$ `if` $C_{a_1} = C_{a_2}$ `then` $\mathcal{F}(P)$ `else` ε, we have that if the condition is satisfied, what transition $[a_1 = a_2]P$ can perform will be simulated by $\mathbb{C} \xrightarrow{*} \mathbb{C}''$ where $\Pi_1(\mathbb{C}) = \mathcal{F}([a_1 = a_2]P)$ and $\Pi_1(\mathbb{C}') = \mathcal{F}(P)$.

(4) For restriction structure $\nu a\ P$: transitions it can execute are based on P. We assume that $P \xrightarrow{\pi_o} P'$ and this transition will be simulated by transitions $\mathbb{C}' \xrightarrow{*} \mathbb{C}''$. By $\mathcal{F}(\nu a\ P) \stackrel{\text{def}}{=} \exists C_a{:}(\mathsf{frame}(C_a) \wedge \mathcal{F}(P))$, we get that what transition $\nu a\ P$ can perform will be simulated by $\mathbb{C} \xrightarrow{*} \mathbb{C}'''$ where $\Pi_1(\mathbb{C}) = \mathcal{F}(\nu a\ P)$ and $\Pi_1(\mathbb{C}''') = \exists C_a{:}(\mathsf{frame}(C_a) \wedge \Pi_1(\mathbb{C}''))$.

(5) For process instance $A\langle a_1, \ldots, a_n\rangle$: transitions it can execute are decided by the definition of $A(x_1, \ldots, x_n)$. According to $A(x_1, \ldots, x_n)$, the transformation of $A\langle a_1, \ldots, a_n\rangle$ only needs to consider two cases, so the proof will be given as follows.

1. Without recursive calls in the definition of A: Since $A(x_1, \ldots, x_n)$ is defined by P_A, we assume $P_A \xrightarrow{\pi_{o_i}} P_i$ ($1 \le i \le m$ and $m \in \mathbb{N}$) and these transitions are simulated by transitions $\mathbb{C}_i \xrightarrow{*} \mathbb{C}_i'$ ($1 \le i \le m$ and $m \in \mathbb{N}$). Since $\mathcal{F}(A\langle a_1, \ldots, a_n\rangle) \stackrel{\text{def}}{=} \{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(P_A)$, we have that what transitions $A\langle a_1, \ldots, a_n\rangle$ can perform will be simulated by $\mathbb{C} \xrightarrow{*} \mathbb{C}_1''$, $\mathbb{C} \xrightarrow{*} \mathbb{C}_2''$,...and $\mathbb{C} \xrightarrow{*} \mathbb{C}_m''$ where $\Pi_1(\mathbb{C}) = \mathcal{F}(A\langle a_1, \ldots, a_n\rangle)$ and $\Pi_1(\mathbb{C}_i'') = \{\overrightarrow{C_a}/\overrightarrow{C_x}\}\Pi_1(\mathbb{C}_i')$.
2. With recursive calls in the definition of A and $A(x_1, \ldots, x_n) \equiv \pi_1^s + \pi_1^{'s} \cdot A\langle x_1, \ldots, x_n\rangle$: Since the transformation employs a recursion procedure to get a given times iteration result of $A\langle a_1, \ldots, a_n\rangle$. The proof will be given by induction on the iteration times j.

Base case For j=0, $A_0\langle a_1,\ldots,a_n\rangle \equiv \{\overrightarrow{a}/\overrightarrow{x}\}\pi_1^s$ where π_1^s denotes $\pi_{1_1}\cdot\cdots\cdot\pi_{1_{n_1}}$, we assume that $\{\overrightarrow{a}/\overrightarrow{x}\}\pi_1^s \xrightarrow{\{\overrightarrow{a}/\overrightarrow{x}\}\pi_{o_{1_1}}} P_1 \ldots P_{n_1-1} \xrightarrow{\{\overrightarrow{a}/\overrightarrow{x}\}\pi_{o_{1_{n_1}}}} 0$ where $\pi_{o_{1_1}}$ and $\pi_{o_{1_{n_1}}}$ are the corresponding observable actions of π_{1_1} and $\pi_{1_{n_1}}$ respectively. Since $\mathcal{F}(A_0\langle a_1,\ldots,a_n\rangle) := \{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^s)$, we clearly have that these transitions will be simulated by $\mathbb{C} \xrightarrow{*} \mathbb{C}_1\cdots\mathbb{C}_{n_1-1} \xrightarrow{*} \mathbb{C}'$ where $\Pi_1(\mathbb{C}) = \{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^s)$ and $\Pi_1(\mathbb{C}') = \varepsilon$.

Induction Suppose that for $\mathcal{F}(A_{j-1}\langle a_1,\ldots,a_n\rangle)$, transitions it can perform will be simulated by $\mathbb{C}_{j-1} \xrightarrow{*} \mathbb{C}'_{j-1}$. Then, for j, we have that $\mathcal{F}(A_j\langle a_1,\ldots,a_n\rangle) := \{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^s) \vee (\{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^{'s}); \mathcal{F}(A_{j-1}\langle a_1,\ldots,a_n\rangle))$. What transitions it can perform will be simulated by $\mathbb{C}_j \xrightarrow{*} \mathbb{C}_1\cdots\mathbb{C}_{n_1-1} \xrightarrow{*} \mathbb{C}'_j$ where $\Pi_1(\mathbb{C}_j) = \{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^s)$ and $\Pi_1(\mathbb{C}'_j) = \varepsilon$ (as in the base case) or $\mathbb{C}_j \xrightarrow{*} \mathbb{C}_{j-1}$ where $\Pi_1(\mathbb{C}_j) = (\{\overrightarrow{C_a}/\overrightarrow{C_x}\}\mathcal{F}(\pi_1^{'s}); \mathcal{F}(A_{j-1}\langle a_1,\ldots,a_n\rangle))$. Therefore, Theorem 2 holds. □

6 Case study

In this section, we employ the rate monotonic scheduling algorithm (RMSA) (Liu and Layland 1973) modeled by p-π. By the transformation $\mathcal{F}$, it will be mapped to an MSVL program and hence verified by techniques of MSVL.

RMSA deals with the schedulability of a set of periodic tasks. Several conclusions have been given in these years, but these methods are undecidable or too intricate. In this paper, we will verify this problem by model checking based on transformation and MSVL. The task set to be considered needs to satisfy the two conditions: (1)The requests of each task are periodic, with constant interval between requests. (2)The deadline constraints specify that each request must be completed before the next request of the same task occurs. The algorithm let the task with shortest executing time have the highest priority.

In this paper, we consider the set of tasks is $\{(A : 2, 8), (B : 3, 10), (C : 4, 12)\}$ where for an element, say $(A : 2, 8)$, the first number 2 is A's executing time and the second 8 is its cycle. The algorithm consists of a scheduling process S, task A TA, task B TB and task C TC. For these tasks, we assume that each task will use the corresponding action $\overline{s_{tart}}\langle t\rangle$ to begin its execution and $\overline{s_{top}}\langle t\rangle$ to complete its execution. The interval between the two actions is the executing time. In addition, we suppose that the waiting deadline of a task equals the difference between the executing time and the cycle. Since A has the shortest executing time, its priority is highest and it may interrupt B and C. Similarly, B may interrupt C. Thus, task A can communicate with the scheduling process to complete its execution and then wait for its deadline, or stop. Unlike task A, task B and C can possibly be interrupted by A so that they have four potential choices: doing the execution, waiting for the deadline, interrupted by other tasks or stopping.

The algorithm is modeled in p-π below. According to the priority, task B will not be executed until A has been executed completely. Thus, A uses $\overline{d_a}$ to trigger the execution of B. Similarly, B executes $\overline{d_b}$ to trigger the execution of C. When the waiting deadline

of A arrives, A employs $\overline{i_b}$ and $\overline{i_c}$ to interrupt B and C respectively. Also, B utilizes $\overline{i_c}$ to interrupt C.

$$
\begin{aligned}
& \nu\ \overrightarrow{n}\ (S\langle\overrightarrow{n_s}\rangle \mid TA\langle\overrightarrow{n_{ta}}\rangle \mid TB\langle\overrightarrow{n_{tb}}\rangle \mid TC\langle\overrightarrow{n_{tc}}\rangle)\\
S(\overrightarrow{n_s}) \ &\stackrel{\text{def}}{=}\ await(s_{start}(m_1).S\langle\overrightarrow{n_s}\rangle) + await(s_{top}(m_2).S\langle\overrightarrow{n_s}\rangle) + \tau\\
TA(\overrightarrow{n_{ta}}) \ &\stackrel{\text{def}}{=}\ await(\overline{s_{start}}\langle t_a\rangle.skip^2.await(\overline{s_{top}}\langle t_a\rangle.await(\overline{d_a}.skip^6.(TA\langle\overrightarrow{n_{ta}}\rangle \mid\\
&\quad await(\overline{i_c})\mid await(\overline{i_b}))))) + \tau\\
TB(\overrightarrow{n_{tb}}) \ &\stackrel{\text{def}}{=}\ await(d_a.await(\overline{s_{start}}\langle t_b\rangle.(skip^3.await(\overline{s_{top}}\langle t_b\rangle.await(\overline{d_b}.TB\langle\overrightarrow{n_{tb}}\rangle)))\\
&\quad + await(i_b.TB\langle\overrightarrow{n_{tb}}\rangle))))) + skip^7.\tau.(TB\langle\overrightarrow{n_{tb}}\rangle\mid await(\overline{i_c})) + await(i_b.\\
&\quad TB\langle\overrightarrow{n_{tb}}\rangle) + \tau\\
TC(\overrightarrow{n_{tc}}) \ &\stackrel{\text{def}}{=}\ await(d_b.await(\overline{s_{start}}\langle t_c\rangle.(skip^4.await(\overline{s_{top}}\langle t_c\rangle.TC\langle\overrightarrow{n_{tc}}\rangle) + await\\
&\quad (i_c.TC\langle\overrightarrow{n_{tc}}\rangle)))) + skip^8.\tau.TC\langle\overrightarrow{n_{tc}}\rangle + await(i_c.TC\langle\overrightarrow{n_{tc}}\rangle) + \tau
\end{aligned}
$$

where $\overrightarrow{n} = s_{start}, s_{top}, t_a, t_b, t_c, d_a, d_b, i_b, i_c$, $\overrightarrow{n_s} = s_{start}, s_{top}$, $\overrightarrow{n_{ta}} = s_{start}, s_{top}, t_a, d_a, i_b, i_c$, $\overrightarrow{n_{tb}} = s_{start}, s_{top}, t_b, d_a, d_b, i_b, i_c$, $\overrightarrow{n_{tc}} = s_{start}, s_{top}, t_c, d_b, i_c$.

By the transformation $\mathcal{F}$ given in Sect. 4, the system will be translated as follows.

$$
\begin{aligned}
& \mathcal{F}(\nu\ \overrightarrow{n}\ (S\langle\overrightarrow{n_s}\rangle \mid TA\langle\overrightarrow{n_{ta}}\rangle \mid TB\langle\overrightarrow{n_{tb}}\rangle \mid TC\langle\overrightarrow{n_{tc}}\rangle)))\\
\equiv\ & \mathsf{prog}_1 \wedge \mathcal{F}(S\langle\overrightarrow{n_s}\rangle \mid TA\langle\overrightarrow{n_{ta}}\rangle \mid TB\langle\overrightarrow{n_{tb}}\rangle \mid TC\langle\overrightarrow{n_{tc}}\rangle)\\
\equiv\ & \mathsf{prog}_1 \wedge (\mathcal{F}(S\langle\overrightarrow{n_s}\rangle) \mid\mid \mathcal{F}(TA\langle\overrightarrow{n_{ta}}\rangle) \mid\mid \mathcal{F}(TB\langle\overrightarrow{n_{tb}}\rangle) \mid\mid \mathcal{F}(TC\langle\overrightarrow{n_{tc}}\rangle)))\\
\equiv^*\ & \mathsf{prog}_1 \wedge (\\
& \quad j_1 := 0;\\
& \quad \mathcal{F}(S_{-1}\langle\overrightarrow{n_s}\rangle) := \varepsilon;\\
& \quad \mathcal{F}(S_0\langle\overrightarrow{n_s}\rangle) := (p_I \wedge \varepsilon);\\
& \quad \mathtt{while}\ (j_1 < N_1)\\
& \quad \mathtt{do}\ (j_1 := j_1 + 1;\\
& \quad \mathcal{F}(S_{j_1}\langle\overrightarrow{n_s}\rangle) := (p_I \wedge \varepsilon) \vee\\
& \quad (\exists C_{m_1} : \mathsf{frame}(C_{m_1}) \wedge (wait\_receive(C_{s_{start}}, C_{m_1}); \mathcal{F}(S_{j_1-1}\langle\overrightarrow{n_s}\rangle))) \vee\\
& \quad (\exists C_{m_2} : \mathsf{frame}(C_{m_2}) \wedge (wait\_receive(C_{s_{top}}, C_{m_2}); \mathcal{F}(S_{j_1-1}\langle\overrightarrow{n_s}\rangle))))\\
& \mid\mid j_2 := 0;\\
& \quad \mathcal{F}(TA_{-1}\langle\overrightarrow{n_{ta}}\rangle) := \varepsilon;\\
& \quad \mathcal{F}(TA_0\langle\overrightarrow{n_{ta}}\rangle) := (p_I \wedge \varepsilon);\\
& \quad \mathtt{while}\ (j_2 < N_2)\\
& \quad \mathtt{do}\ (j_2 := j_2 + 1;\\
& \quad \mathcal{F}(TA_{j_2}\langle\overrightarrow{n_{ta}}\rangle) := (p_I \wedge \varepsilon) \vee\\
& \quad (wait\_send(C_{s_{start}}, C_{t_a}); skip^2; wait\_send(C_{s_{top}}, C_{t_a}); wait\_send(C_{d_a}, nil);\\
& \quad skip^6; (\mathcal{F}(TA_{j_2-1}\langle\overrightarrow{n_{ta}}\rangle) \mid\mid wait\_send(C_{i_b}, nil) \mid\mid wait\_send(C_{i_c}, nil))))\\
& \mid\mid j_3 := 0;\\
& \quad \mathcal{F}(TB_{-1}\langle\overrightarrow{n_{tb}}\rangle) := \varepsilon;\\
& \quad \mathcal{F}(TB_0\langle\overrightarrow{n_{tb}}\rangle) := (p_I \wedge \varepsilon);\\
& \quad \mathtt{while}\ (j_3 < N_3)\\
& \quad \mathtt{do}\ (j_3 := j_3 + 1;
\end{aligned}
$$

$\mathcal{F}(TB_{j_3}\langle\overrightarrow{n_{tb}}\rangle) := (p_I \wedge \varepsilon) \vee$
$(wait\_receive(C_{d_a}, nil);\ wait\_send(C_{s_{start}}, C_{t_b});\ ((skip^3;\ wait\_send(C_{s_{stop}}, C_{t_b});$
$wait\_send(C_{d_b}, nil);\ \mathcal{F}(TB_{j_3-1}\langle\overrightarrow{n_{tb}}\rangle)) \vee (wait\_receive(C_{i_b}, nil);\ \mathcal{F}(TB_{j_3-1}$
$\langle\overrightarrow{n_{tb}}\rangle))))) \vee (skip^7;\ (p_I \wedge \varepsilon);\ (\mathcal{F}(TB_{j_3-1}\langle\overrightarrow{n_{tb}}\rangle)||wait\_send(C_{i_c}))) \vee$
$(wait\_receive(C_{i_b}, nil);\ \mathcal{F}(TB_{j_3-1}\langle\overrightarrow{n_{tb}}\rangle)))$
$||\ j_4 := 0;$
$\mathcal{F}(TC_{-1}\langle\overrightarrow{n_{tc}}\rangle) := \varepsilon;$
$\mathcal{F}(\mathrm{TC}_0\langle\overrightarrow{n_{tc}}\rangle) := (p_I \wedge \varepsilon);$
`while` $(j_4 < N_4)$
`do` $(j_4 := j_4 + 1;$
$\mathcal{F}(\mathrm{TC}_{j_4}\langle\overrightarrow{n_{tc}}\rangle) := (p_I \wedge \varepsilon) \vee$
$(wait\_receive(C_{d_b}, nil);\ wait\_send(C_{s_{start}}, C_{t_c});\ ((skip^4;\ wait\_send(C_{s_{stop}}, C_{t_c});$
$\mathcal{F}(\mathrm{TC}_{j_4-1}\langle\overrightarrow{n_{tc}}\rangle))) \vee (wait\_receive(C_{i_c}, nil);\ \mathcal{F}(\mathrm{TC}_{j_4-1}\langle\overrightarrow{n_{tc}}\rangle))))) \vee (skip^8;\ (p_I \wedge \varepsilon);$
$\mathcal{F}(\mathrm{TC}_{j_4-1}\langle\overrightarrow{n_{tc}}\rangle))) \vee (wait\_receive(C_{i_c}, nil);\ \mathcal{F}(\mathrm{TC}_{j_4-1}\langle\overrightarrow{n_{tc}}\rangle)))))$
(where prog_1 denotes $\exists C_{s_{start}}, C_{s_{stop}}, C_{t_a}, C_{t_b}, C_{t_c}, C_{d_a}, C_{d_b}, C_{i_b}, C_{i_c}$: frame
$(C_{s_{start}}, C_{s_{stop}}, C_{t_a}, C_{t_b}, C_{t_c}, C_{d_a}, C_{d_b}, C_{i_b}, C_{i_c})$.)

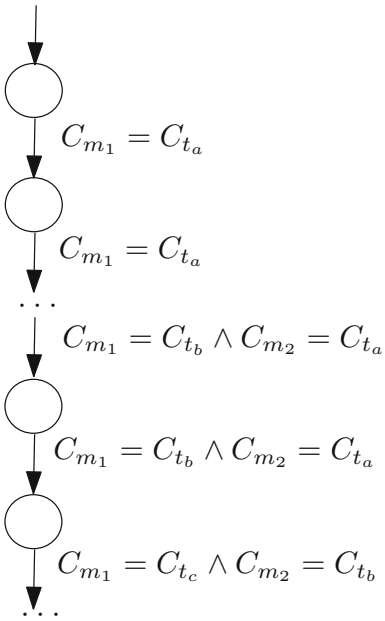
where $N_i (1 \leq i \leq 4)$ is a constant integer.

For process instances $S\langle\overrightarrow{n_s}\rangle$, $TA\langle\overrightarrow{n_{ta}}\rangle$, $TB\langle\overrightarrow{n_{tb}}\rangle$ and $TC\langle\overrightarrow{n_{tc}}\rangle$, there are recursive calls in their definitions. Thus, their transformations utilize the second transformation rule of the process instance.

According to the obtained MSVL programs, we will verify properties of time-dependent systems modeled in p-π by means of MSVL. To this end, we employ a modeling, simulation and verification tool for MSVL. Actually, the tool can work in the following three modes: (1) in the modeling mode, given the MSVL program p of a system, the state space of the system can implicitly be given as an NFG (Normal Form Graph) of p (Duan et al. 2008a); (2) with the simulation mode, an execution path of the NFG of the system is presented as the output with respect to operational semantics of MSVL (Yang and Duan 2008); (3) under the verification mode, given a system model described by an MSVL program p, and a property specified by a PPTL (Propositional PTL) formula ϕ, it can automatically be checked whether the system satisfies the property or not (whether or not $p \rightarrow \phi$ is valid) (Duan and Tian 2008). The main idea of the verification is that, whether or not the system p satisfies the property ϕ can equivalently be checked by evaluating whether or not $\neg(p \rightarrow \phi) \equiv p \wedge \neg\phi$ is unsatisfiable. Therefore, the satisfiability of a formula in the form of $p \wedge \neg\phi$ is checked by constructing the NFG of $p \wedge \neg\phi$, and then detecting whether or not there exist paths in the NFG. If there are no paths, the property is satisfied. On the other hand, the counterexample can be given, which means the system does not satisfy the property. In this article, we focus on the verification mode.

In the modeling mode of MSVL, the state space of the algorithm discussed before can be created and presented as an NFG in Fig. 2. In the NFG, an edge denotes the current state assignment for the variables, e.g., $C_{m_1} = C_{ta}$ means that the currently executing task is TA and $C_{m_2} = C_{tb}$ shows that the latest completed task is TB. Notice that, in this NFG, we focus on the main variables, ignoring those without influence on our discussions.

Fig. 2 NFG of algorithm



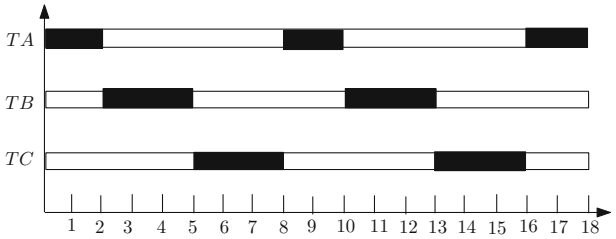


Fig. 3 Execution of algorithm

The execution status of the algorithm is demonstrated in Fig. 3.

For this algorithm, we pay attention to whether a given group of tasks are schedulable or not. To this end, we verify the property that "all of the tasks will completely be executed without interrupting in their cycles". For convenience of discussion, propositions p_a and q_a are employed to denote $C_{m_1} = C_{ta}$ (TA is executed in the current state) and $C_{m_2} = C_{ta}$ (TA is the latest completed task) respectively. The usage of p_b, p_c, q_b and q_c are similar. Therefore, this property can be specified by $(\Diamond q_a \wedge \Diamond q_b \wedge \Diamond q_c)^*$ in PPTL where $\Diamond$ means *sometimes* and $\Diamond p \stackrel{\text{def}}{=} \mathsf{true}\ ;\ p$. With the verification mode of MSVL, we add the following code

$$
\begin{aligned}
< /define\ \ p_a : C_{m_1} = C_{ta}; define\ \ q_a : C_{m_2} = C_{ta};\\
define\ \ p_b : C_{m_1} = C_{tb}; define\ \ q_b : C_{m_2} = C_{tb};\\
define\ \ p_c : C_{m_1} = C_{tc}; define\ \ q_c : C_{m_2} = C_{tc};\\
(som\ \ q_a\ \ and\ \ som\ \ q_b\ \ and\ \ somq_c)^{\#} / >
\end{aligned}
$$

Fig. 4 Verification result 1

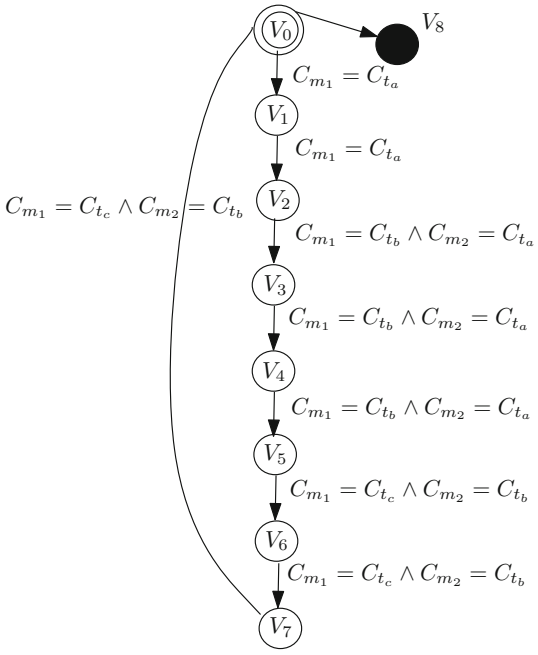


Fig. 5 Verification result 2

to the beginning of the MSVL code for the verification of time-dependent properties, then run the code with model checker. The resulting NFG is produced as shown in Fig. 4. Obviously, there exist infinite paths where node $V_i\,(0 \leq i \leq 7)$ appears finitely often and q_c never appears so that we can obtain that TC has never been completely executed. Hence, the property cannot be satisfied.

In order to make sure that this group of tasks can be schedulable, we have to adjust the cycles of the tasks. Consider the adjusted set $\{(A : 2, 9), (B : 3, 10), (C : 4, 12)\}$. For this set, we do the modelling, transformation and verification again. An empty NFG with no edges is produced as shown in Fig. 5. The result means that the formula is unsatisfiable so that the algorithm for the adjusted tasks satisfies the property.

In addition, we can also verify other important time-dependent properties for the algorithm, such as mutual exclusivity, priority, etc. Due to the limit of space, we omit the concrete verification of these properties here.

7 Conclusion

In this paper, We propose a structural transformation $\mathcal{F}$ from p-π processes to MSVL programs. This enables us to make use of the theories and techniques of MSVL to analyze p-π processes. More precisely, p-π is able to model, simulate and verify its

processes by means of MSVL. In the future, we will further investigate the possibility for transforming MSVL programs into p-π processes so that a tight relationship between MSVL and p-π could be established. In addition, supporting tools for transformation between p-π processes and MSVL programs are needed.

Acknowledgments This research is supported by the National Program on Key Basic Research Project of China (973 Program) Grant no. 2010CB328102, National Natural Science Foundation of China under Grant no. 61133001, 61202038, 61272117, 61272118, 61322202 and 91218301.

References

Bergstra J, Klop J (1985) Algebra of communicating processes with abstraction. J Theor Comput Sci 37:77–121

Duan Z (1996) An extended interval temporal logic and a framing technique for temporal logic programming. Ph.D Thesis, University of Newcastle Upon Tyne

Duan Z (2006) Temporal logic and temporal logic programming. Science Press, Beijing

Duan Z, Tian C (2008) A unified model checking approach with projection temporal logic. In: Proceedings of ICFEM 2008. LNCS 5256, pp 167–186

Duan Z, Koutny M, Holt C (1994) Projection in temporal logic programming. In: Proceedings of logic programming and automatic reasoning. LNAI 822, Springer, pp 333–344

Duan Z, Tian C, Zhang L (2008a) A decision procedure for propositional projection temporal logic with infinite models. Acta Inform 45(1):43–78

Duan Z, Yang X, Koutny M (2008b) Framed temporal logic programming. Sci Comput Programm 70(1):31–61

Hoare C (1985) Communicating sequential processes. Prentice-Hall, London

Liu C, Layland J (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J Assoc Comput Mach 20(1):46–61

Luo L, Duan Z (2012) An extended π-calculus. Proc CNSI 2012:632–637

Luo L, Duan Z (2013) A transformation from p-π to msvl. In: Proceedings of ICTAC 2013. LNCS 8049, pp 267–281

Milner R (1980) A calculus of communicating systems. Lecture Notes in Computer Science 92. Springer, Heidelberg

Milner R (1999) Communicating and mobile systems: the π-calculus. Cambridge University Press, Cambridge

Milner R, Parrow J, Walker D (1992) A calculus of mobile processes. Inf Comput 100:1–77

Sangiorgi D, Walker D (2002) The π-calculus: a theory of mobile processes. Cambridge University Press, Cambridge

Tarski A (1955) A lattice-theoretical fixpoint theorem and its applications. Pac J Math 5:285–309

Winskel G (1993) The formal semantics of programming languages: an introduction. Cambridge University Press, Cambridge

Yang X, Duan Z (2008) Operational semantics of framed tempura. doi:10.1016/j.jlap.2008.08.001

Yang X, Duan Z, Ma Q (2010) Automatic verification of finite state concurrent system using temporal logic specification. Math Struct Comput Sci 20(5):865–914