ICOT Technical Report: TR-0879

TR-0879

Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE

by
N. Uchihira & S. Honiden

June, 1994

© Copyright 1994-6-30 ICOT, JAPAN ALL RIGHTS RESERVED

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5

Institute for New Generation Computer Technology

Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE

Naoshi UCHIHIRA and Shinichi HONIDEN

Systems & Software Engineering Lab.
R & D Center, TOSHIBA Corporation,
70, Yanagi-cho, Saiwai-ku, Kawasaki 210, JAPAN.
e-mail: uchi@ssel.toshiba.co.jp
phone: +81-44-548-5474
fax: +81-44-533-3593

Abstract

In this paper, we examine "program adjustment", a formal and practical approach to developing correct concurrent programs, by automatically adjusting an imperfect program to satisfy given constraints. A concurrent program is modeled by a finite state process, and program adjustment to satisfy temporal logic constraints is formalized as the synthesis of an arbiter process which partially serializes target (i.e. imperfect) processes to remove harmful nondeterministic behaviors. Compositional adjustment is also proposed for large-scale compound target processes, using process equivalence theory. We have developed a computer-aided programming environment on the parallel computer Multi-PSI, called MENDELS ZONE, that adopts this compositional adjustment. Adjusted programs can be compiled into the kernel language (KL1) and executed on Multi-PSI.

KEY WORDS: Concurrent Program, Program Synthesis, Program Adjustment, CCS, Temporal Logic, Büchi Automaton, Finite State Process, Bisimulation, Programming Environment.

1 Introduction

1.1 Motivation

As practical parallel and distributed computing gradually spreads into the industry, there is an increasing demand for programmers who design concurrent programs. It is not easy for ordinary programmers to produce correct and efficient concurrent programs. In particular debugging concurrent programs requires a great deal of labor. Some kind of computer-aided concurrent programming environment is seriously needed.

The difficulty of concurrent program debugging is mainly due to its nondeterministic behavior. We classify nondeterminism into the following 3 types.

- **Intended nondeterminism:** Nondeterministic behaviors which the programmer intends to implement.
- **Harmful nondeterminism:** Nondeterministic behaviors which the programmer does not intend to implement and does not expect.
- **Persistent nondeterminism:** Nondeterministic behaviors which have no effect on the results.

For example, Fig.1 shows a simple concurrent program "*Seat Booking*", where two processes read/write a shared memory "*Seat*" to reserve one seat. This program has the 3 types of nondeterministic behaviors.

Intended nondeterminism The following nondeterministic behaviors θ_1 and θ_2 derive different results: P_1 can book the seat ($Status_1 = ok$) in θ_1, but cannot ($Status_1 = ng$) in θ_2. However both are correct (intended behaviors).

- $\theta_1 = l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_1 \rightarrow m_2 \rightarrow m_5$
  Result: $Status_1 = ok, Seat = occupied, Status_2 = ng$.
- $\theta_2 = m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow m_5 \rightarrow l_1 \rightarrow l_2 \rightarrow l_5$
  Result: $Status_1 = ng, Seat = occupied, Status_2 = ok$.

Harmful nondeterminism The following nondeterministic behavior θ_3 derives an incorrect result (double booking). So, this program has harmful nondeterminism.

- $\theta_3 = l_1 \rightarrow m_1 \rightarrow l_2 \rightarrow m_2 \rightarrow l_3 \rightarrow m_3 \rightarrow l_4 \rightarrow m_4 \rightarrow l_5 \rightarrow m_5$
  Result: $Status_1 = ok, Seat = occupied, Status_2 = ok$.

Persistent nondeterminism The following two nondeterministic behaviors have a same result because l_1(write in $Status_1$) and m_1(write in $Status_2$) are independent actions each other. We call such a situation *persistent*.

- $\theta_4 = \mathrm{l}_1 \rightarrow \mathrm{m}_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $Status_1 = ok, Seat = occupied, Status_2 = ng.$
- $\theta_5 = \mathrm{m}_1 \rightarrow \mathrm{l}_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $Status_1 = ok, Seat = occupied, Status_2 = ng.$

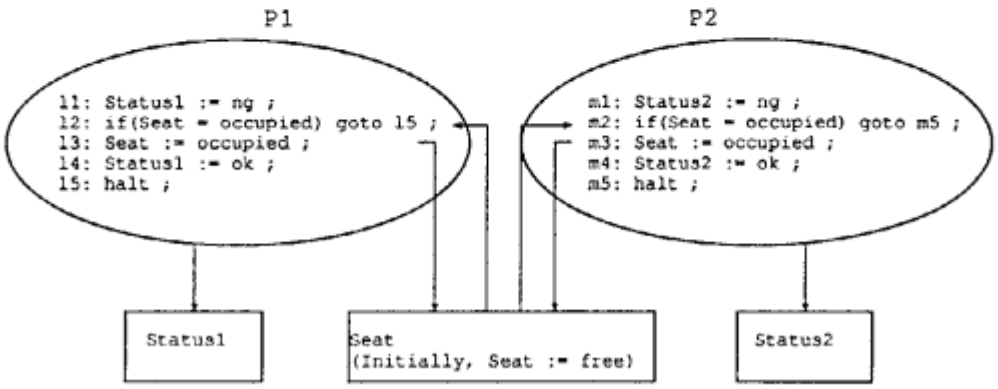


Figure 1: An example of a concurrent program

In our observation of concurrent program development, a programmer first tries to design and implement processes so as to maximize concurrency, which may include 3 types of nondeterminism. He then often finds harmful nondeterministic behaviors in testing and debugs them by partially serializing the critical sections which interfere each other using synchronization mechanisms (e.g. semaphores). Bugs due to harmful nondeterministic behaviors often account for a considerable part of all timing bugs.

We will show that the debugging processes for harmful nondeterministic behaviors can be mechanically supported using formal methods. It can be also regarded as a practical application of program synthesis techniques to program modification in debugging.

1.2 Overview of Main Results

We propose "program adjustment" which automatically adjusts (debugs) an imperfect program to satisfy given constraints. Here, we consider only timing constraints

for concurrent programs that can be specified by temporal logic. In this context, "an imperfect program" is regarded as a program which is functionally correct but may be imperfect in its timing. We call such a program an FCTI program (Functionally-Correct Temporally-Imperfect program).

A concurrent program is modeled with the finite state process [12] , which can specify the finite state transition system with liveness conditions. It can not only represent the transition systems in CCS [13], but also Büchi automata [14]. A target FCTI program is compositionally constructed from several finite state processes with the composition operator "|"(ex. $P = (P_{11} \mid P_{12}) \mid (P_{21} \mid P_{22})$ in Fig.2(a)).

Basic Adjustment Program adjustment (Basic Adjustment) means to adjust an FCTI program to satisfy given constraints by adding an arbiter process which is synchronized with and restricts the behavior of the FCTI program (Fig.2(b)). The arbiter partially serializes the FCTI program to remove harmful nondeterministic alternatives which do not satisfy given constraints. We will show an algorithm to synthesize an arbiter process C_f automatically.

Input: An FTCI program P.

Input: Temporal logic constraints f.

Output: A arbiter process C_f such that $P \mid C_f$ satisfies f.

Compositional Adjustment When a target program becomes large, the arbiter synthesis may cause computing cost explosion. Therefore, we propose compositional adjustment, in which local arbiters are synthesized in each composition step. For example, an adjusted program with local arbiters C_0, C_1, and C_2 is shown as follows (Fig.2(c)).

$$P' = (P_{11} \mid P_{12} \mid C_1) \mid (P_{21} \mid P_{22} \mid C_2) \mid C_0$$

In each composition step, the reduction of the finite state process, based on process equivalence theory, can ease computing cost explosion. We introduce a new process equivalence relation ($\pi\tau\omega-$bisimulation) to manipulate liveness properties because the traditional weak bisimulation equivalence used in CCS cannot. $\pi\tau\omega-$bisimulation is used to reduce a finite state process to a smaller and equivalent one in the compositional adjustment.

It is more feasible for ordinary programmers to adopt the program adjustment approach compared to other methods which synthesize complete programs from (temporal logic) specifications [9][10][26]. The reasons are as follows.

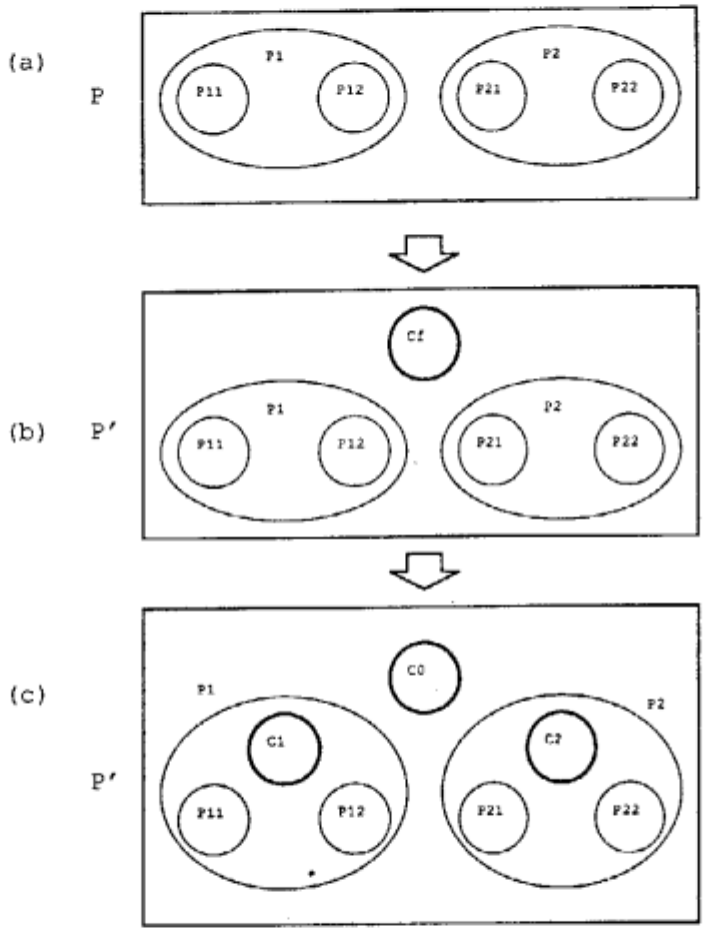


Figure 2: Process Composition (a), Basic (b), and Compositional (c) Adjustment

- It is not very difficult for ordinary programmers to produce an FCTI concurrent program, which satisfies at least the functional requirements. A more difficult task is to design and debug the timing of such programs.
- Many bugs are derived from harmful nondeterministic alternatives.
- It is easy for ordinary programmers to specify timing constraints, such as deadlock-free and starvation-free constraints, as compared with implementing them.

MENDELS ZONE In order to confirm the feasibility of program adjustment, we have developed a concurrent programming environment, MENDELS ZONE, which adopts the compositional adjustment in cooperation with the verification. In MENDELS ZONE, the programmer first finds existing bugs by the verification step, then adjusts the program to remove the bugs by the adjustment step (Fig.3).

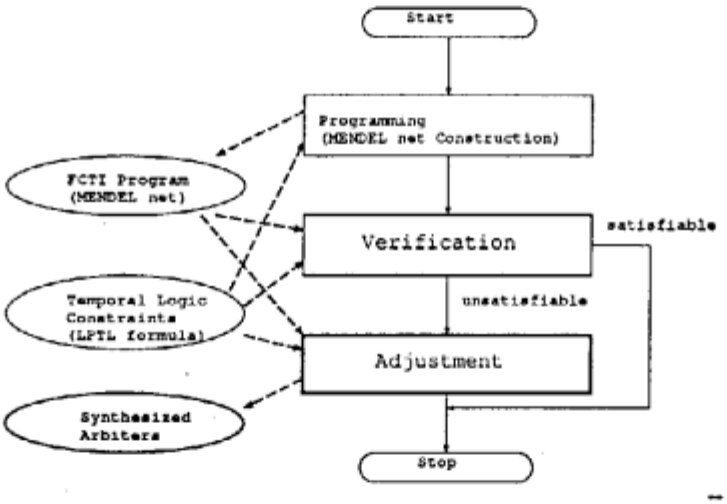


Figure 3: Verification and Adjustment in MENDELS ZONE

1.3 Significance of this paper

1. **Theoretical Aspect:** The traditional CCS framework (composition and equivalence) is not adequate for finite state processes with the liveness conditions (i.e. Büchi automata). Therefore, we introduce new composition and

equivalence for finite state processes which can preserve liveness properties. These techniques are essential to the basic and compositional adjustment.

2. **Practical Aspect:** We introduce the concept of "program adjustment" into the concurrent programming which is based on the formal method but feasible for ordinary programmers, and have implemented the programming environment (MENDELS ZONE) adopting the program adjustment to show its effectiveness.

1.4 Organization of the paper

The remainder of the paper is organized as follows. Section 2 defines Finite State Processes (FSP) and their equivalence relation and composition operator. Basic and compositional adjustment of FSP is described in Section 3. An overview of MENDELS ZONE is briefly shown and its compositional adjustment is explained in Section 4. Finally, Section 5 shows a simple and nontrivial example of program adjustment, followed by the conclusion in Section 6.

2 FINITE STATE PROCESSES

The basic model for concurrent programs is the finite state process [12] , which can specify the finite state transition system with *liveness conditions.* First, we define a Finite State Process (FSP) and an equivalence relation for FSPs. Then, several operators (composition, relabeling, and reduction) on FSPs are introduced and their properties are shown.

2.1 Finite State Processes

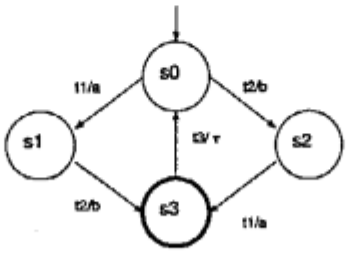
Definition 1 (Finite State Process) *A Finite State Process (FSP) is a seven-tuple* $P = (S, A, L, \delta, \pi, s_0, F)$, *where:*

- S *is a finite set of states,*
- A *is a finite set of actions,*
- L *is a finite set of synchronization labels,*
- $\delta : S \times A \rightarrow S \cup \{\perp\}$ *is a* **deterministic** *transition function* ($\delta(s,t) = \perp$ *means the action* $t \in A$ *is disabled in the state* $s \in S$),
- $\pi : A \rightarrow (L \cup \{\tau\})$ *is a labeling function,* (τ *is an invisible internal action),*
- $s_0 \in S$ *is an initial state, and*

- $F \subset S$ *is a set of designated states.*

■

Example 1 (Finite State Process) $P = (\{s_0, s_1, s_2, s_3\}, \{t_1, t_2, t_3\}, \{a, b\}, \delta, \pi, s_0, \{s_3\})$ *is a finite state process where* $\delta(s_0, t_1) = s_1, \delta(s_0, t_2) = s_2, \delta(s_1, t_2) = s_3, \delta(s_2, t_1) = s_3, \delta(s_3, t_3) = s_0, \pi(t_1) = a, \pi(t_2) = b, \pi(t_3) = \tau$. *(Fig.4)* ■



NOTE: action/label; a bold circle means a designated state.

Figure 4: Finite State Process

To begin with, we introduce several notations. Let X be a set. The set of all finite sequences over X, including the empty sequence ε, is denoted by X^*. If there is no empty sequence ε, the set is denoted by X^+. The set of all infinite sequences over X is denoted by X^ω; ω means "infinitely many". X^∞ is defined by $X^\infty = X^* \cup X^\omega$.

For a sequence $\theta \in X^\infty$, $\theta[i]$ means the i-th element in θ; $\theta(k)$ means the prefix subsequence $\theta[1]\theta[2]...\theta[k]$ of θ, and $\mid \theta \mid$ the length of θ.

Let $P = (S, A, L, \delta, \pi, s_0, F)$ be an FSP. A transition function can be extended such that $\delta : S \times A^* \rightarrow S \cup \{\perp\}$, i.e., $\delta(s, \theta a) \stackrel{def}{=} \delta(\delta(s, \theta), a)$. Note, $\delta(s, \varepsilon) = s$. Since a transition function is deterministic, a current state can be uniquely determined from an initial state and an action sequence. We call an action sequence a *behavior*. Similarly, we can extend a labeling function such that $\pi : A^* \rightarrow (L \cup \{\tau\})^*$, i.e., $\pi(\theta) = \pi(\theta[1])\pi(\theta[2])...\pi(\theta[|\ \theta\ |])$. In addition, $\hat{\pi}(\theta)$ is defined as the sequence gained by deleting all occurrences of τ from $\pi(\theta)$. The set of reachable states from a state s in P is defined as $R_P(s) \stackrel{def}{=} \{s' \in S \mid \exists \theta \in A^*.s' = \delta(s, \theta)\}$ and $R_P^+(s) \stackrel{def}{=} \{s' \in S \mid \exists \theta \in A^+.s' = \delta(s, \theta)\}$. Also, the set of all possible action sequences of P is defined as $L(P) \stackrel{def}{=} \{\theta \in A^* \mid \delta(s_0, \theta) \neq \perp\}$, and the set of all

possible label sequences is defined as $L_\pi(P) \stackrel{def}{=} \{\hat{\pi}(\theta) \in L^* \mid \theta \in L(P)\}$. Since interest is in the infinite behavior of an FSP, we introduce a set of infinite action sequences $L_\omega(P) \subset (A^\omega \cup A^*\{\Delta\}^\omega)$ where Δ means *deadlock*:

$$L_\omega(P) \stackrel{def}{=} \left\{ \begin{array}{l} \{\theta \in A^\omega \mid 1 \leq \forall k.\delta(s_0,\theta(k)) \neq \bot\} \cup \\ \{\theta \in A^*\{\Delta\}^\omega \mid \exists k.(\left\{ \begin{array}{l} 1 \leq \forall i \leq k.\delta(s_0,\theta(i)) \neq \bot \text{ and} \\ \forall a \in A.\delta(\delta(s_0,\theta(k)),a) = \bot \text{ and} \\ \theta[j] = \Delta \text{ for } \forall j > k \end{array} \right.)\} \end{array} \right.$$

$L_\omega(P)$ is an extension of $L(P)$ into a set of infinite action sequences where if $\theta \in L(P)$ is a deadlock sequence (i.e., an inevitably finite sequence), then θ is represented as $\theta\Delta^\omega \in L_\omega(P)$.

$L_\omega^{fair}(P) \subset L_\omega(P)$ is defined as $L_\omega^{fair}(P) \stackrel{def}{=} \{\theta \mid \theta \in L_\omega(P)$ under the fairness condition$\}$ where the fairness condition means whenever a behavior θ infinitely often passes through some state s, every action a enabled at s must appear infinitely often on θ (i.e., if $s = \delta(s_0,\theta(i))$ for infinitely many i and $\delta(s,a) \neq \bot$, then $s = \delta(s_0,\theta(j))$ and $\theta[j+1] = a$ for infinitely many j).

Finally, $L(P)/A'$ is introduced by definition: $L(P)/A' \stackrel{def}{=} \{\theta' \mid \exists\theta \in L(P).\forall i.(\theta'[i] = \varepsilon$ if $\theta[i] \in A'$, otherwise $\theta'[i] = \theta[i])\}$ Intuitively, $L(P)/A'$ consists of action sequences of P in which all elements of $A' \subset A$ are deleted.

An FSP is a transition system with liveness conditions. In an FSP, liveness conditions are represented by designated nodes that indicate satisfiable behavior of an FSP as follows.

Definition 2 (Satisfiable Behavior) *Let $P = (S, A, L, \delta, \pi, s_0, F)$ be an FSP. $\theta \in A^\omega$ is a satisfiable behavior, if $\delta(s_0,\theta(k)) \in F$ for infinitely many $k \geq 1$. $L_b(P) \subset A^\omega$ is defined as a set of all satisfiable behaviors on P.* ■

Note that a satisfiable behavior corresponds to an accepting run of Büchi automaton.

Definition 3 (Completeness of FSP) *Let $P = (S, A, L, \delta, \pi, s_0, F)$ be an FSP. P is complete if $\forall s \in R_P(s_0).\exists s' \in R_P^+(s)$ and $s' \in F$.* ■

A state $s \in R_P(s_0)$, having no path to designated nodes from s, is called an *unsatisfiable state*. A behavior reaching an unsatisfiable state is called an *inevitably unsatisfiable behavior*.

Lemma 1 *If an FSP P is complete, then $L_\omega^{fair}(P) \subset L_b(P)$.* ■

This lemma means that if P is complete, then a random transition over P leads to a satisfiable behavior.

2.2 Equivalence of Finite State Processes

We now introduce the notion of $\pi\tau\omega-$bisimulation equivalence that is an extension of Milner's weak bisimulation equivalence [15][13] . $\pi\tau\omega-$bisimulation equivalence was originally developed for compositional verification [5] . In this paper, it is used to reduce an FSP to a smaller and equivalent one in compositional adjustment.

Definition 4 ($\tau\omega-$divergence) *Let $P = (S, A, L, \delta, \pi, s_0, F)$ be an FSP. $s \in S$ is $\tau\omega-$divergent ($s\uparrow$) if $\forall n > 0.\exists s' \in S.\exists \theta \in A^*.(\mid \theta \mid = n, \hat{\pi}(\theta) = \varepsilon$ and $s' = \delta(s, \theta))$.* ■

Definition 5 ($\pi\tau\omega-$bisimulation Equivalence) *Let $P_1 = (S_1, A_1, L_1, \delta_1, \pi_1, s_{01}, F_1)$ and $P_2 = (S_2, A_2, L_2, \delta_2, \pi_2, s_{02}, F_2)$ be FSPs. P_1 and P_2 are $\pi\tau\omega-$bisimulation equivalent ($P_1 \approx_{\pi\tau\omega} P_2$), if there is a binary relation $R \subset S_1 \times S_2$, such that $(s_{01}, s_{02}) \in R$, and $\forall s_1 \in S_1.\forall s_2 \in S_2.(s_1, s_2) \in R \iff$*

- *$s_1 \in F_1$ iff $s_2 \in F_2$,*
- *$s_1\uparrow$ iff $s_2\uparrow$,*
- *$\forall t_1 \in A_1.\forall s_1' \in S_1.($ if $s_1' = \delta_1(s_1, t_1)$ then*
  $\exists \theta \in A_2^*.\exists s_2' \in S_2.\hat{\pi}_1(t_1) = \hat{\pi}_2(\theta), s_2' = \delta_2(s_2, \theta),$ and $(s_1', s_2') \in R)$,
- *$\forall t_2 \in A_2.\forall s_2' \in S_2.($ if $s_2' = \delta_2(s_2, t_2)$ then*
  $\exists \theta \in A_1^*.\exists s_1' \in S_1.\hat{\pi}_2(t_2) = \hat{\pi}_1(\theta), s_1' = \delta_1(s_1, \theta),$ and $(s_1', s_2') \in R)$.

■

$\pi\tau\omega-$bisimulation is extended so that it can discriminate designated states and divergence, which cannot be discriminated by weak bisimulation (the weak bisimulation ignores divergences, i.e., τ-loops and τ-circles). The following lemma is derived from these discrimination abilities.

Lemma 2 *If P_1 is complete and $P_1 \approx_{\pi\tau\omega} P_2$, then P_2 is also complete.* ■

Definition 6 (Reduction) *For a given FSP $P = (S, A, L, \delta, \pi, s_0, F)$, a reduction of P, $red(P) = (S_r, A_r, L_r, \delta_r, \pi_r, s_{r0}, F_r)$, is an FSP such that $P \approx_{\pi\tau\omega} red(P)$ and $\mid S_r \mid \leq \mid S \mid$.* ■

The smallest $red(P)$ is constructed effectively by the relational coarsest partitioning algorithm [16][12] such that all states of P that are $\pi\tau\omega-$bisimilar to each other are brought together into a single state of $red(P)$.

2.3 Operators on Finite State Processes

Concurrent programs are constructed as a composition of several FSPs that are synchronized with each other. The composition and relabeling operators for FSPs are introduced and their important properties (substitutivity and reflectivity) are shown.

Definition 7 (Composition Operator) *For* $P_1 = (S_1, A_1, L_1, \delta_1, \pi_1, s_{10}, F_1)$ *and* $P_2 = (S_2, A_2, L_2, \delta_2, \pi_2, s_{20}, F_2)$, *a composition* $P = P_1 \mid P_2$ *is defined as follows.*

$P = (S_1 \times S_2 \times \{0,1\}^2, (A_1 \cup \{idle\}) \times (A_2 \cup \{idle\}), L_1 \cup L_2, \delta, \pi, (s_{10}, s_{20}, 0, 0), F)$, *where*

- $\delta : (S_1 \times S_2 \times \{0,1\}^2) \times (A_1 \cup \{idle\}) \times (A_2 \cup \{idle\}) \rightarrow (S_1 \times S_2 \times \{0,1\}^2) \cup \{\perp\}$ *such that*
  $\delta((s_1, s_2, f_1, f_2), (a_1, a_2)) =$

$$
\begin{cases}
(\delta_1(s_1,a_1), \delta_2(s_2,a_2), f_1', f_2'), \text{ when } \pi_1(a_1) = \pi_2(a_2) \neq \tau, \text{ and } f_1 = f_2 = 1, \\
\quad \text{where} \left\{ \begin{array}{ll} f_i' = 1 & \text{if } \delta_i(s_i,a_i) \in F_i, \\ f_i' = 0 & \text{otherwise}, \end{array} \right\} (\text{for each } i = 1,2) \\
(\delta_1(s_1,a_1), \delta_2(s_2,a_2), f_1', f_2'), \text{ when } \pi_1(a_1) = \pi_2(a_2) \neq \tau, \text{ and } (f_1 = 0 \vee f_2 = 0), \\
\quad \text{where} \left\{ \begin{array}{ll} f_i' = 1 & \text{if } \delta_i(s_i,a_i) \in F_i \vee f_i = 1, \\ f_i' = 0 & \text{otherwise}, \end{array} \right\} (\text{for each } i = 1,2) \\
(\delta_1(s_1,a_1), s_2, f_1', 0), \text{ when } \pi_1(a_1) \notin (L_1 \cap L_2), a_2 = idle, \text{ and } f_1 = f_2 = 1, \\
\quad \text{where} \left\{ \begin{array}{ll} f_1' = 1 & \text{if } \delta_1(s_1,a_1) \in F_1, \\ f_1' = 0 & \text{otherwise}, \end{array} \right. \\
(\delta_1(s_1,a_1), s_2, f_1', f_2), \text{ when } \pi_1(a_1) \notin (L_1 \cap L_2), a_2 = idle, \text{ and } (f_1 = 0 \vee f_2 = 0), \\
\quad \text{where} \left\{ \begin{array}{ll} f_1' = 1 & \text{if } \delta_1(s_1,a_1) \in F_1 \vee f_1 = 1, \\ f_1' = 0 & \text{otherwise}, \end{array} \right. \\
(s_1, \delta_2(s_2,a_2), 0, f_2'), \text{ when } \pi_2(a_2) \notin (L_1 \cap L_2), a_1 = idle, \text{ and } f_1 = f_2 = 1, \\
\quad \text{where} \left\{ \begin{array}{ll} f_2' = 1 & \text{if } \delta_2(s_2,a_2) \in F_2, \\ f_2' = 0 & \text{otherwise}, \end{array} \right. \\
(s_1, \delta_2(s_2,a_2), f_1, f_2'), \text{ when } \pi_2(a_2) \notin (L_1 \cap L_2), a_1 = idle, \text{ and } (f_1 = 0 \vee f_2 = 0), \\
\quad \text{where} \left\{ \begin{array}{ll} f_2' = 1 & \text{if } \delta_2(s_2,a_2) \in F_2 \vee f_2 = 1, \\ f_2' = 0 & \text{otherwise}, \end{array} \right. \\
\perp, \text{ when otherwise}.
\end{cases}
$$

- $\pi : (A_1 \cup \{idle\} \times A_2 \cup \{idle\}) \rightarrow L_1 \cup L_2 \cup \{\tau\}$ *such that*

$$
\begin{cases}
\pi((a_1,a_2)) = \pi_1(a_1) = \pi_2(a_2) & \text{if } a_1 \in A_1 \text{ and } a_2 \in A_2, \\
\pi((a_1, idle)) = \pi_1(a_1) & \text{if } a_1 \in A_1, \\
\pi((idle, a_2)) = \pi_2(a_2) & \text{if } a_2 \in A_2,
\end{cases}
$$

- *and* $F = \{(s_1, s_2, f_1, f_2) \mid s_1 \in S_1, s_2 \in S_2, f_1 = f_2 = 1\}$.

■

Remark that processes are synchronized at actions *with same labels* in the above process composition. This composition is similar to composition of CCS[13] except for its treatment of designated nodes. The following relabeling operators are used to relabel actions so that actions which are synchronized in composition have same labels.

Definition 8 (Relabeling Operator) *For* $P = (S, A, L, \delta, \pi, s_0, F)$ *and a relabeling function* $f : L \to L' \cup \{\tau\}$, $P' = P[f]$ *is defined as follows.*

$$P' = (S, A, L', \delta, \pi', s_0, F), \text{ where } \begin{cases} \pi'(a) = f(\pi(a)) & \text{if } \pi(a) \neq \tau, \\ \pi'(a) = \tau & \text{if } \pi(a) = \tau. \end{cases}$$

■

Example 2 (Composition and Relabeling)

- $P_1 = (\{s_0, s_1, s_2\}, \{t_1, t_2, t_3, t_4, t_5\}, \{a_1, b_1, c\}, \delta_1, \pi_1, s_0, \{s_1\})$ *where*
  $\delta_1(s_0, t_1) = s_1, \delta_1(s_0, t_2) = s_2, \delta_1(s_1, t_3) = s_2, \delta_1(s_2, t_4) = s_1, \delta_1(s_1, t_5) = s_1, \pi_1(t_1) = a_1, \pi_1(t_2) = b_1, \pi_1(t_3) = b_1, \pi_1(t_4) = a_1, \pi_1(t_5) = c.$
- $P_2 = (\{s_0, s_1, s_2\}, \{t_1, t_2, t_3, t_4, t_5\}, \{a_2, b_2, d\}, \delta_2, \pi_2, s_0, \{s_2\})$ *where*
  $\delta_2(s_0, t_1) = s_1, \delta_2(s_0, t_2) = s_2, \delta_2(s_1, t_3) = s_2, \delta_2(s_2, t_4) = s_1, \delta_2(s_2, t_5) = s_2, \pi_2(t_1) = a_2, \pi_2(t_2) = b_2, \pi_2(t_3) = b_2, \pi_2(t_4) = a_2, \pi_2(t_5) = d.$
- *relabeling functions:* $f_i(a_i) = a, f_i(b_i) = b$, *and* $f_i(l) = l$ *for other labels* $l \in \{c, d\}$ *(for each i=1,2).*
- $P_1[f_1] \mid P_2[f_2] = (\{s_0, s_1, s_2, s_3, s_4\}, \{(t_1, t_1), (t_2, t_2), (t_3, t_3), (t_4, t_4), (t_5, idle), (idle, t_5)\}, \{a, b, c, d\}, \delta, \pi, s_0, \{s_3, s_4\})$ *where*
  $\delta(s_0, (t_1, t_1)) = s_1, \delta(s_0, (t_2, t_2)) = s_2, \delta(s_1, (t_3, t_3)) = s_3, \delta(s_1, (t_5, idle)) = s_1, \delta(s_2, (t_4, t_4)) = s_4, \delta(s_2, (idle, t_5)) = s_2, \delta(s_3, (t_4, t_4)) = s_1, \delta(s_3, (idle, t_5)) = s_2, \delta(s_4, (t_3, t_3)) = s_2, \delta(s_4, (t_5, idle)) = s_1, \pi((t_1, t_1)) = a, \pi((t_2, t_2)) = b, \pi((t_3, t_3)) = b, \pi((t_4, t_4)) = a, \pi((t_5, idle)) = c, \pi((idle, t_5)) = d.$

(Fig.5) ■

Definition 9 (Projection) *Let* P_1 *and* P_2 *be FSPs. A left projection* $L(P_1 \mid P_2) \downarrow left$ *is defined as* $L(P_1 \mid P_2) \downarrow left \stackrel{def}{=} \{\theta_1/\{idle\} \mid \exists \theta \in L(P_1 \mid P_2).\theta[i] = (\theta_1[i], \theta_2[i])\}$. *Similarly, a right projection* $L(P_1 \mid P_2) \downarrow right$ *is defined. In the same way, projections of* L_ω, L_ω^{fair}, *and* $L_\flat$ *are defined.* ■

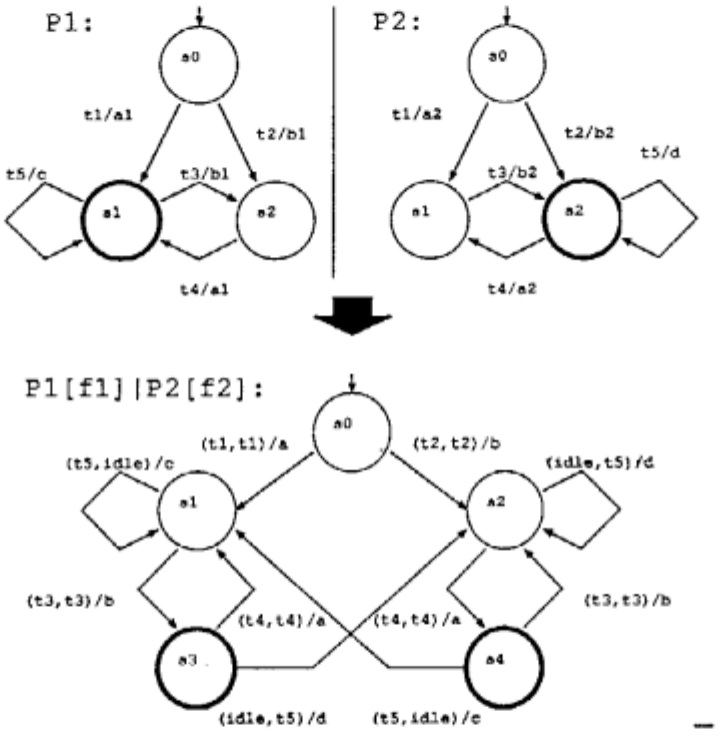


Figure 5: Composition and Relabeling

Lemma 3 (Reflectivity) *Let P_1 and P_2 be FSPs. If $P = P_1 \mid P_2$, then $L_b(P) \downarrow left \subset L_b(P_1)$ and $L_b(P) \downarrow right \subset L_b(P_2)$.* ■

Lemma 4 (Substitutivity) *$\pi\tau\omega-$bisimulation equivalence is preserved by composition and relabeling; that is, if $P \approx_{\pi\tau\omega} Q$, then $\forall R.(P \mid R \approx_{\pi\tau\omega} Q \mid R)$, and $\forall f.(P[f] \approx_{\pi\tau\omega} Q[f])$.* ■

Reflectivity and substitutivity are used in the basic adjustment and the compositional adjustment, respectively. These adjustments are described in the next section.

3 PROGRAM ADJUSTMENT

This section proposes program adjustment of FSPs. First, we show that a temporal logic constraint f can be transformed to an equivalent FSP P_f. For an FTCI process P and a temporal logic constraint f, $P \mid P_f$ is a composed process in which P's behaviors against f are disabled by P_f (i.e., safety properties are satisfied). However, $P \mid P_f$ is not necessarily complete (i.e., liveness properties may not be satisfied). Program adjustment means to make $P \mid P_f$ complete by adding arbiter process C (i.e., the adjusted program $= P \mid P_f \mid C$).

3.1 Temporal Logic

The constraints for concurrent programs (safety properties and liveness properties) are specified by temporal logic. Safety properties include admissible partial ordering of actions (i.e., transition firing), and liveness properties include deadlock and starvation about actions.

Definition 10 (LPTL) Syntax *Linear time propositional temporal logic (LPTL) formulas are built from:*

- *a set of all atomic propositions: $Prop = \{p_1, p_2, p_3, ..., p_n\}$,*
- *boolean connectives: $\wedge$, $\neg$, and*
- *temporal operators: $\bigcirc$("next"), U("until").*

The formation rules are as follows.

- *An atomic proposition $p \in Prop$ is a formula.*
- *If f_1 and f_2 are formulas, so are $f_1 \wedge f_2$, $\neg f_1$, $\bigcirc f_1$, $f_1 U f_2$.*

Semantics *The operators intuitively have the following meanings.*

- $\neg$:*NOT;*
- $\wedge$:*AND;*
- $\bigcirc f$ *(read next f):f is true for the next state;*
- $f_1 U f_2$ *(read* f_1 *until* f_2*):*f_1 *is true until* f_2 *becomes true and* f_2 *will eventually become true.*

The precise semantics are given as the Kripke structure[9] .

■

We use $\Diamond f$ ("eventually f") as an abbreviation for $true\ U\ f$ and $\Box f$ ("always f") as an abbreviation for $\neg\Diamond\neg f$. Also, $f_1 \vee f_2$ and $f_1 \Rightarrow f_2$ represent $\neg(\neg f_1 \wedge \neg f_2)$ and $\neg f_1 \vee f_2$, respectively. Here, we assume *a single event condition* under which only one atomic proposition is true at any moment.

Theorem 1 *Given an LPTL formula f under a single event condition, one can build an FSP* $P_f = (S, A, L, \delta, \pi, s_0, F)$ *such that L corresponds to a set of atomic propositions of f, and* $L_b(P_f)$ *is exactly the set of behaviors whose label sequences satisfy the formula f.*

(Proof) It is a restriction of a general theorem [19] .

■

Remark that a label sequence of a satisfiable behavior in P_f corresponds to a model of LPTL formula.

Example 3 (Temporal Logic Constraints) *Let a label set be* $L = \{a_1, a_2\}$.

- *(1)* $\Box\Diamond(a_1 \vee a_2)$*: Either* a_1 *or* a_2 *must infinitely often occur.*
- *(2)* $\Box(a_1 \supset \bigcirc\Box(\neg a_2))$*: Whenever* a_1 *occurs, then* a_2 *must never occur.*

FSPs which are generated from (1) and (2) are shown in Fig.6.

In the context of the following program adjustment, we restrict temporal logic formulas so that P_f is **deterministic** with regard to synchronization labels. In this case, some formulas, such as $\Diamond\Box a$, which are translated to nondeterministic one, become not available. These formulas are suitable for verification, but not for adjustment (synthesis) because the arbiter cannot look ahead future behaviors as indicated by Pnueli and Rosner[30][31].

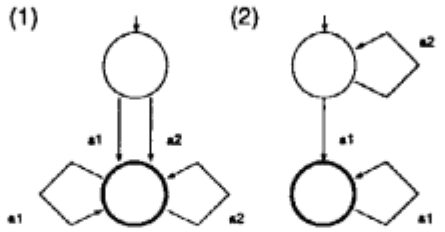


Figure 6: FSPs P_f of Temporal Logic Constraints

3.2 Basic Adjustment

As temporal logic constraints f can be translated to an FSP P_f, we will show how to make an FSP $P = P_f \mid P_0$ complete for the target FCTI program P_0. In the following explanation, we assume that the target FSP P has already composed with P_f (i.e., $P = P_f \mid ....$), and do not mention P_f explicitly.

Problem 1 (Basic Adjustment)

Input: *An FSP* $P = (S, A, L, \delta, \pi, s_0, F)$ *(We assume* $P = P_f \mid ...$*).*

Output: *A maximally permissive FSP* $C = (S_c, A_c, L_c, \delta_c, \pi_c, s_{0c}, F_c)$ *such that* $P \mid C$ *is complete.*

"C is maximally permissive" means that for every C' *if* $P \mid C'$ *is complete then* $L(P \mid C') \subset L(P \mid C)$. ■

The arbiter, C, restrains the target FSP P from falling into unsatisfiable states by eliminating harmful observable transitions.

Algorithm 1 (Single Arbiter Synthesis)

(Step 0) $P' := P$.

(Step 1) *Find a set of unsatisfiable states* $S_u \subset S'$ *in* $P' = (S', A', L, \delta', \pi', s'_0, F')$. *If there are no unsatisfiable states, go to Step 4.*

(Step 2) *Construct a pseudo-arbiter* C' *from* P' *as follows. At first,* τ *– closure* C_τ *is defined as*
$C_\tau(s, a) \stackrel{def}{=} \{s' \mid \exists\theta.(s' = \delta(s, \theta), \hat{\pi}(\theta) = a)\}$ *for* $\forall s \in S'$ *and* $\forall a \in L \cup \{\varepsilon\}$,

$C_\tau(S_{sub}, a) \stackrel{def}{=} \bigcup_{s \in S_{sub}} C_\tau(s, a)$ *for* $\forall S_{sub} \subset S'$ *and* $\forall a \in L \cup \{\varepsilon\}$,
then
$C' = (S'_c, A'_c, L, \delta'_c, \pi'_c, C_\tau(s'_0, \varepsilon), S'_c)$, *where* $S'_c = 2^{S'}$, $A'_c = \{t_a \mid a \in L\} \cup \{t_s \mid s \in S'\}$, *and for* $\forall a \in L$, $\forall s' \in S'_c$,

- $\delta'_c(s', t_a) = C_\tau(s', a) \in S'_c$ *if* $C_\tau(s', a) \cap S_u = \emptyset$,
- $\delta'_c(s', t_a) = \perp$ *if* $C_\tau(s', a) \cap S_u \neq \emptyset$,
- $\delta'_c(s', t_{s'}) = s'$,

and $\pi'_c(t_a) = a$ *and* $\pi'_c(t_{s'}) = \tau$ *for* $\forall a \in L, \forall s' \in S'_c$.
Remark that "$\delta'_c(s', t_a) = \perp$ *if* $C_\tau(s', a) \cap S_u \neq \emptyset$*" means elimination of all behaviors which cannot be distinguished from inevitably unsatisfiable behaviors by a label observer.*

(Step 3) $P' := P' \mid C'$, *and return to Step 1.*

(Step 4) *Let the final pseudo-arbiter* C', *which is generated after applying Step 1 - Step 3 repeatedly, be the arbiter* C.

If C is empty (i.e., all behaviors are eliminated), C is called *unrealizable*; otherwise, C is called *realizable.*

Theorem 2 (Main Theorem) *If an FSP* $C = (S_c, A_c, L_c, \delta_c, \pi_c, s_{0c}, F_c)$ *is realizable for a given FSP* $P = (S, A, L, \delta, \pi, s_0, F)$ *in the above algorithm, then* $P \mid C$ *is complete and* C *is maximally permissive.*

(Sketch of proof) During Step 1 - Step 3, all inevitably unsatisfiable behaviors are eliminated in the final P'. *Therefore,* P' *is complete. Since the transition function of* C' *is deterministic about its labels,* C' *restrains no satisfiable behavior of* P. *Therefore* $P \mid C$ *is complete and* C *is maximally permissive.* ■

Corollary 1

$$L_\omega^{fair}(P \mid C) \downarrow left \subset L_b(P \mid C) \downarrow left \subset L_b(P)$$

(Proof) This proof is derived from Lemma 1 and Lemma 3 with Theorem 2. ■

This corollary assures that P, adjusted by C, satisfies its liveness constraints, whenever its behaviors are made by random transitions over states. Remark that an arbiter is effective in case $L_\omega^{fair}(P) \subset L_b(P)$ does not hold (i.e., P has harmful nondeterministic behaviors).

Example 4 (A single arbiter synthesis) *Fig.7 shows a simple single arbiter synthesis. In the target process P, only $\theta = t_3t_6t_7$ is an inevitably unsatisfiable behavior. Since $\{t_3t_6t_7, t_3t_4\}$ is a set of behaviors which cannot be distinguished from θ (i.e. have the same label sequence "ab"), t_4 and t_7 are eliminated. From the remainder, the arbiter C can be constructed.*

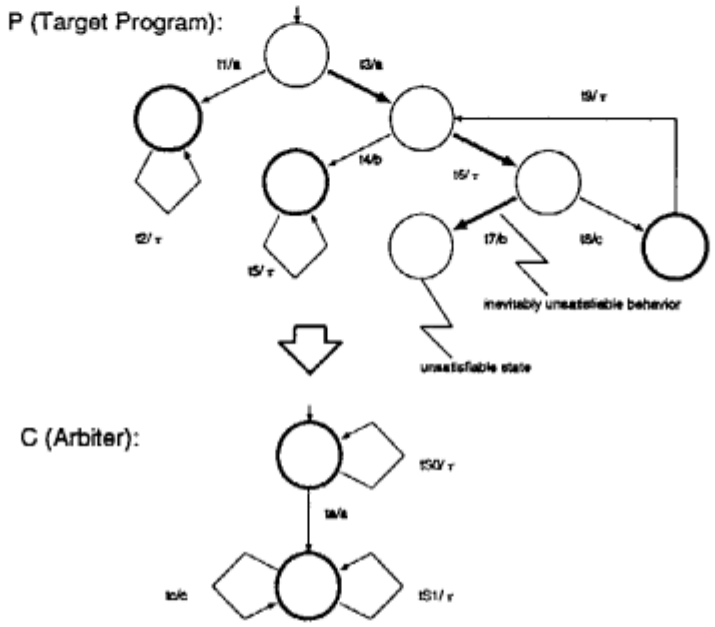


Figure 7: Single Arbiter Synthesis

3.3 Compositional Adjustment

When a target program that is composed hierarchically with many processes becomes very large, the arbiter synthesis may cause the following problems.

1. The synthesis results in computing cost explosion,
2. A single arbiter is too restrictive to control the whole program precisely.

Therefore, we propose compositional adjustment, in which local arbiters are synthesized in each composition step. The reduction of an FSP can ease the computing cost explosion in each step.

Theorem 3 *If $P_1 \approx_{\pi\tau\omega} P_2$, then C is an arbiter of P_1 iff C is an arbiter of P_2.*
(Proof) From Lemma 2 and Lemma 4, $C \mid P_1$ is complete iff $C \mid P_2$ is complete.
■

Corollary 2 *If C is an arbiter of red(P), then C is also an arbiter of P.*
■

Algorithm 2 (Compositional Arbiter Synthesis) *For simplicity, we explain compositional adjustment for the following target program that is constructed by two-level composition (Fig.2(c)). This algorithm can be extended easily to arbitrary target programs.*

- *Target Program:*

$$(P_{11}[h_{11}] \mid P_{12}[h_{12}])[h_1] \mid (P_{21}[h_{21}] \mid P_{22}[h_{22}])[h_2]$$

where P_{11}, P_{12}, P_{21}, and P_{22} are FSPs, and $h_{11}, h_{12}, h_{21}, h_{22}, h_1$ and h_2 are relabeling functions.

- *Temporal Logic Constraints:*
  f_1, f_2, f_0 are temporal logic constraints for each composition level.

The compositional arbiter synthesis is done in a bottom-up way (Fig. 8).

(Step 1) *Low level arbiters C_1 and C_2 are synthesized for subprocesses $P_{11}[h_{11}] \mid P_{12}[h_{12}] \mid P_{f_1}$ and $P_{21}[h_{21}] \mid P_{22}[h_{22}] \mid P_{f_2}$, respectively. We denote $P_1 \stackrel{def}{=} (C_1 \mid P_{11}[h_{11}] \mid P_{12}[h_{12}] \mid P_{f_1})[h_1]$ and $P_2 \stackrel{def}{=} (C_2 \mid P_{21}[h_{21}] \mid P_{22}[h_{22}] \mid P_{f_2})[h_2]$.*

(Step 2) *Reduced subprocesses $red(P_1)$ and $red(P_2)$ are made from P_1 and P_2.*

(Step 3) *A top level arbiter C_0 is synthesized for a target process $red(P_1) \mid red(P_2) \mid P_{f_0}$.*

Corollary 2 assures that reduction preserves all information necessary for each local arbiter synthesis. The reduction in each step can cut down the synthesis cost. As the ratio of internal actions in the process increases, so does the effectiveness of the reduction. Note that it is possible to synthesize directly a single arbiter C' for the target programs. However, C' is too restrictive because it has less controllable actions compared with local arbiters, and its synthesis cost is more expensive without reduction. Process reduction by weak bisimulation equivalence has been already proposed and shown its effectiveness in compositional verification by Clarke et. al. [27]. However, the reduction preserving liveness properties by $\pi\tau\omega$-bisimulation is our original work.

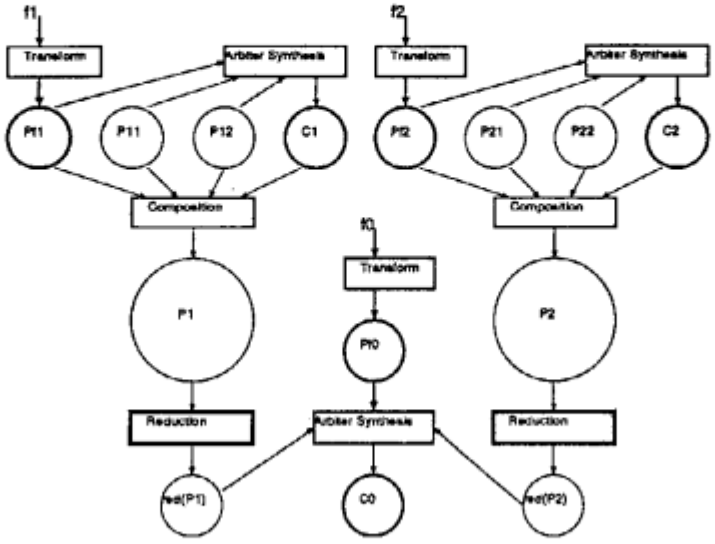


Figure 8: Compositional Arbiter Synthesis

4 MENDELS ZONE

4.1 OVERVIEW

MENDELS ZONE is a programming environment for concurrent programs. The target concurrent programming language, MENDEL [23], is based on a high-level Petri net. It is translated into the concurrent logic programming language KL1 [24] and executed on the parallel machine Multi-PSI [25]. MENDEL is regarded as a user-friendly macro language of KL1, whose purpose is similar to A'UM [17] and AYA [18] . However, MENDEL is more convenient for programmers to use to design a state-transition-based distributed system. MENDELS ZONE supports (1) synthesis of MENDEL atomic processes [7], (2) graphical process interconnection [6], (3) compositional adjustment of interconnected MENDEL processes based on theories described in Section 3, and (4) performance design[8]. This adjustment procedure, which needs relatively high computing power, is implemented by KL1 and executed on Multi-PSI to achieve an effective speedup.

4.2 MENDEL NET

MENDEL is a concurrent programming language based on a high-level Petri net. If a programmer constructs a program using only the MENDELS ZONE's graphic editor shown in Fig.9, he does not have to learn the detailed syntax of MENDEL. He is required only to know a graphical representation of the high-level Petri net, called MENDEL net. Therefore, we omit an explanation of MENDEL itself. MENDEL net is extended from a Petri net in the following aspects.

- Modularity is introduced. A module of MENDEL net represents a process.
- Synchronous (i.e., hand-shake) communication between processes is introduced, in addition to asynchronous (i.e., dataflow) communication.
- Each transition can have an additional enable condition, which must be satisfied when the transition fires, and an additional action, which is executed when it fires. Both are written by KL1.

MENDEL net is graphically represented like a Petri net (Fig.10). The basic conventions are as follows.

- Each "place" is represented by a circle.
- Each "transition" is represented by a square.
- Each process is represented by enclosing places and transitions belonging to the process with a line.
- A "synchronous (hand-shake) communication" is represented by a dotted line between transitions.
- An "asynchronous (dataflow) communication" is represented by an arrow between a transition and a place.

Our program adjustment method is only applicable to finite state programs. When program adjustment is applied, the target MENDEL net is restricted to being a bounded one without asynchronous communications, which can be translated into FSPs. Furthermore, KL1 codes attached to transitions are ignored in the adjustment.

4.3 MENDEL NET CONSTRUCTION

A programmer can construct a MENDEL net using the graphic editor and the program library as follows.

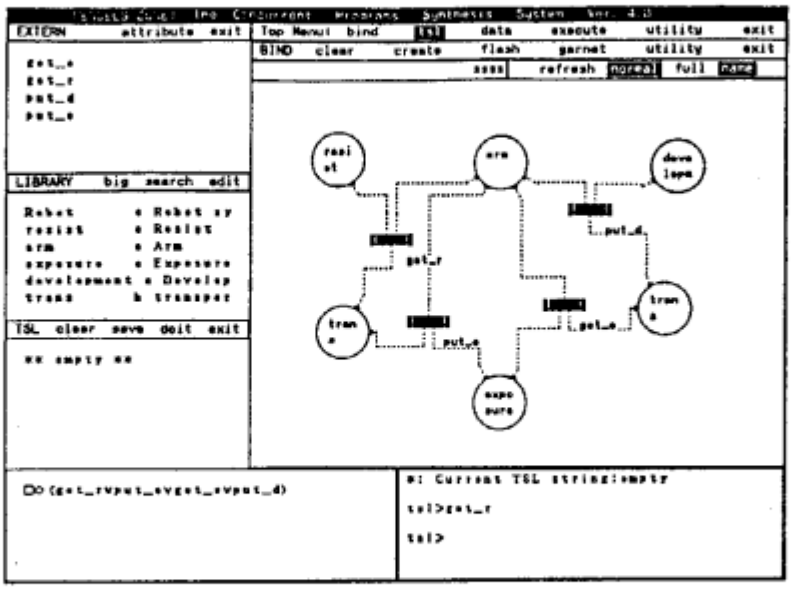


Figure 9: MENDELS ZONE

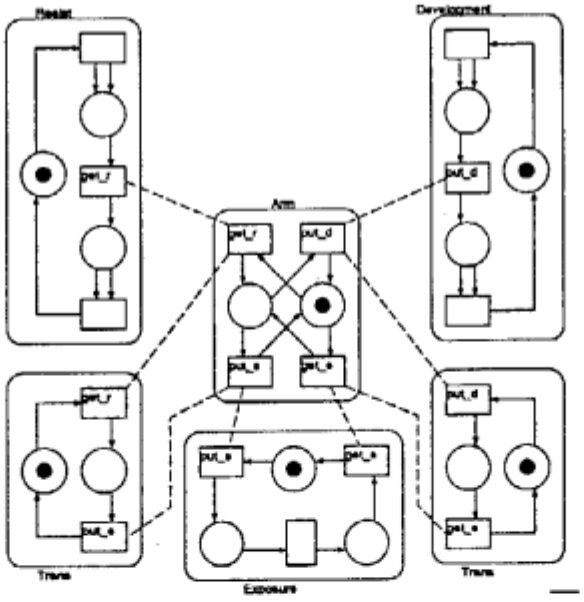


Figure 10: MENDEL net

- (Step 1) Construct atomic MENDEL processes basically by software reuse [1]. If the library has no suitable reusable MENDEL processes, MENDELS ZONE can synthesize it from a given algebraic specification[7] . It is also possible for the programmer to construct the atomic MENDEL process by himself using the graphic editor.
- (Step 2) Interconnect MENDEL processes with communication links using the graphic editor to make a new compound MENDEL process. A large-scale program can be constructed in this compositional way.

Constructed programs are FCTI because a programmer reuses programs whose possible behaviors he may not fully understand; so communication links may be incomplete.

4.4 MENDEL NET VERIFICATION AND ADJUSTMENT

After constructing an FCTI MENDEL net, the programmer specifies safety and liveness properties that must be satisfied by MENDEL net. These properties are specified by temporal logic.

The verification and adjustment procedure (Fig.3) in MENDELS ZONE is as follows.

1. The programmer can give an LPTL formula for a MENDEL net of each compound process.
2. MENDELS ZONE checks whether a MENDEL net satisfies a given LPTL formula by the model checking method for LPTL [20] .
3. When it does not satisfy the LPTL formula, the adjustment method is invoked.

4.5 COMPILATION TO KL1 AND EXECUTION

The adjusted MENDEL program is compiled into a KL1 program, which can be executed on Multi-PSI. The programmer can check visually that the adjusted program satisfies his expectation. If not, he should consider two types of bugs: (1) Bugs in the temporal logic constraints, and (2) Bugs in the KL1 code attached to transitions (i.e., its enable conditions and additional actions), which are ignored in translating to FSP.

5 EXAMPLE: The Machine Control Program

In this example we synthesize a single arbiter using MENDELS ZONE. The problem may be stated informally as follows. The target program must be designed to control machines which cooperatively process (i.e., etch) printed circuit boards (Fig.11). The coating machine applies resist to boards. The exposure machine exposes boards to the light. The development machine develops boards. The arm machine moves boards from one machine to another. The target program is composed of 6 processes (*Resist*, *Exposure*, *Development*, *Arm*, and *Trans* $\times$ 2) which control corresponding machines. *Trans* represents board transportation. Each process is displayed as a MENDEL net, shown in Fig.10. With no arbiter, this system is FCTI because it falls into deadlock when an action label sequence of Arm "$get\_r \rightarrow put\_e \rightarrow get\_r$" occurs. We give the following temporal logic constraints:

$$f = \Box\Diamond F(get\_r \vee put\_e \vee get\_e \vee put\_d)$$

which means Arm never falls into deadlock. An arbiter C is synthesized as follows: first, FSPs representing 6 subprocesses are relabeled by relabeling functions $f_r, f_e, f_d, f_a, f_{t1}$, and f_{t2}, and are reduced, and FSP P_f (Fig.12) representing temporal logic constraints f is generated. The target process P (Fig.13) is composed from these FSPs (including P_f). Finally, the arbiter C shown in Fig.14 is synthesized from P, according to Algorithm 1. We can see that the adjusted program "$C \mid P_f \mid Resist[f_r] \mid Exposure[f_e] \mid Development[f_d] \mid Arm[f_a] \mid Trans[f_{t_1}] \mid Trans[f_{t_2}]$" satisfies the above constraints. Figure 15 shows this adjustment visually in MENDELS ZONE. You can see the target program (left) is automatically transformed into the adjusted one (right), where the arbiter process is unfold in the top level.

6 CONCLUSIONS AND RELATED WORKS

Program adjustment consists of partially synthesizing programs to remove bugs that are due to harmful nondeterministic behaviors. In the proposed framework, program adjustment is defined as the synthesis of arbiter processes which control target processes with synchronization to satisfy their temporal logic constraints. We have had some experience in state-transition-based software construction, using compositional adjustment in MENDELS ZONE. For example we have constructed a control software for a power plant (about 4,300 steps) and evaluated MENDELS ZONE[28].

Our previous works[1][3][4] had proposed program synthesis methods based on temporal logic. However, these methods generated a global state transition graph

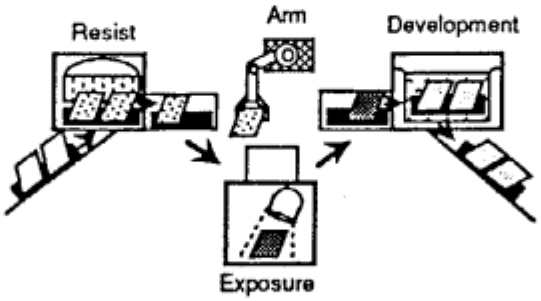


Figure 11: Machine for Processing Printed Circuit Boards

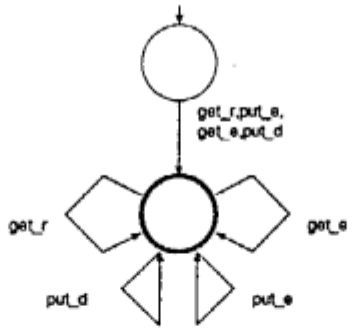


Figure 12: FSP P_f for LPTL formula f

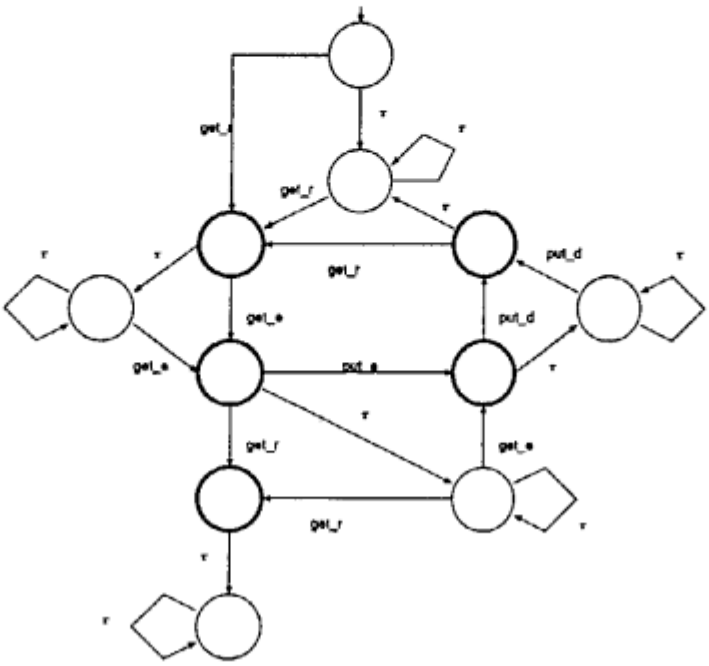


Figure 13: Target Process P(displaying only labels)

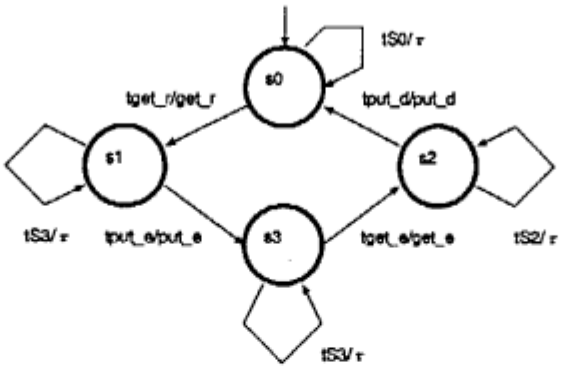
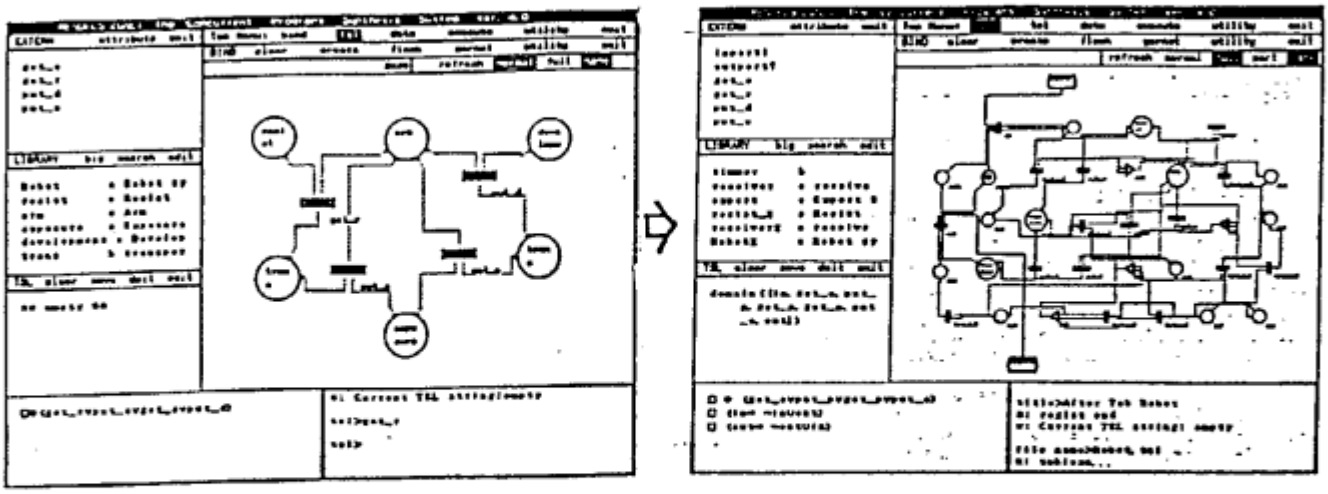


Figure 14: Synthesized Arbiter C



Target Program Adjusted Program

Figure 15: Program Adjustment in MENDELS ZONE

based on the assumption that all process actions are visible (not internal) and controllable. This assumption is restrictive, and the state transition graph often becomes huge, and its generation is expensive since it cannot be done compositionally. In this paper, we introduce a CCS-like compositional framework to achieve compositional adjustment utilizing process reduction. Abadi, Lamport, and Wolper[21] proposed a compositional program synthesis using the CCS-like compositional framework, where failure equivalence is adopted instead of our $\pi\tau\omega-$bisimulation equivalence. However, their approach is a top-down program refinement, which differs from our bottom-up program adjustment approach. From another view, arbiter synthesis can be regarded as a control problem of discrete event systems which are well surveyed by Ramadge and Wonham[22] . However, while these works mainly consider safety properties, they showed no compositional synthesis methods satisfying liveness constraints.

The concurrency control of database transactions [29] is much related to the program adjustment. Both are intended to remove harmful nondeterminism. The program adjustment can be regarded as the extended concurrency control applied to compositional (hierarchical) concurrent programs.

ACKNOWLEDGMENTS

This research has been supported by ICOT. We would like to thank Ryuzou Hasegawa of ICOT for his encouragement and support. We are also grateful to Sadakazu Watanabe and Kazuo Matsumura of the Systems & Software Engineering Laboratory, TOSHIBA Corporation, for providing continuous support.

References

[1] N. Uchihira, et al., Concurrent Program Synthesis with Reusable Components Using Temporal Logic, COMPSAC'87 (1987).

[2] S. Honiden, et al., An Application of Structural Modeling and Automated Reasoning to Concurrent Program Design, 22nd HICSS (1989).

[3] N. Uchihira, et al., Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse, 23nd HICSS (1990).

[4] N.Uchihira and S.Honiden, Verification and synthesis of concurrent programs using Petri nets and temporal logic, Trans. IEICE, Vol.E73, No.12 (1990).

[5] N. Uchihira, PQL: Modal Logic for Compositional Verification of Concurrent Programs (in Japanese), Trans. IEICE Vol.J75-DI, No.2 (1992).

[6] N.Uchihira, et.al., A Petri-Net-Based Programming Environment and Its Design Methodology for Cooperating Discrete Event Systems, IEICE Trans. Vol.E75-A, No.10 (1992).

[7] S.Honiden, A.Ohsuga, N.Uchihira, An integrating environment to put formal specifications into practical use in real-time systems, Proc. IWSSD'91 (1991).

[8] S.Honiden, N.Uchihira, K.Itoh, An Application of Artificial Intelligence to Prototyping Process in Performance Design for Real-Time Systems, ESEC'91, LNCS 550 (1991), also to appear at IEEE Trans. on SE.

[9] Z.Manna and P.Wolper, Synthesis of Communicating Processes from Temporal Logic Specification, ACM Trans. Program. Lang. & Syst., Vol. 6, No. 1 (1984).

[10] E.A.Emerson, E.M.Clarke, Using Branching Time Temporal Logic To Synthesize Synchronization Skeletons, Science of Computer Programming 2 (1982).

[11] J.W. de Bakker, et al. (ed.), Stepwise Refinement of Distributed Systems, REX Workshop, LNCS 430 (1989).

[12] P. C. Kanellakis, S. A. Smolka, CCS Expressions, Finite State Processes and Three Problems of Equivalence, Information and Computation 86 (1990).

[13] R.Milner, Communication and Concurrency, Prentice Hall (1989).

[14] J.R.Büchi, A decision method in restricted second order arithmetic, Proc. Internat. Congr. Logic, Method. and Philos. Sci. (1960), Stanford University Press (1962).

[15] D.Park, Concurrency and automata on infinite sequences, Lecture Notes in Computer Science Vol. 104, Springer-Verlag (1981).

[16] R. Paige, R.E.Tarjan, Three Partition Refinement Algorithms, SIAM J. Comput. 16, No.6 (1987).

[17] K. Yoshida, T. Chikayama, A'UM -Stream-Based Concurrent Object-Oriented Language -, FGCS88 (1988).

[18] K.Suzaki and T.Chikayama, AYA: Process-Oriented Concurrent Programming Language on KL1 (in Japanese), KL1 Programming Workshop'91 (1991).

[19] P.Wolper, et al., Reasoning about Infinite Computation Paths, IEEE Proc. of 24th FOCS (1983).

[20] M.Y.Vardi, P.Wolper, An Automata-Theoretic Approach To Automatic Program Verification, LICS86 (1986).

[21] M. Abadi et al., Realizable and Unrealizable Specifications of Reactive Systems, 16th ICALP (1989).

[22] P.J.Ramadge and W.M.Wonham, The control of discrete event systems, Proc. IEEE, Vol.77, No.1 (1989).

[23] S.Honiden,N.Uchihira,T.Kasuya, MENDEL: Prolog Based Concurrent Object Oriented Language, COMPCON'86 (1986).

[24] T.Chikayama, et.al., Overview of the Parallel Inference Machine Operating System (PIMOS), Proc.FGCS'88 (1988).

[25] K.Taki, The FGCS Computing Architecture, ICOT TR-460 (1989).

[26] A.Pnueli, R.Rosner, Distributed Reactive Synthesis are Hard to Synthesis, 31th FOCS (1990).

[27] E.M.Clarke, E.E.Long, K.L.McMillan, Compositional Model Checking, Proc. 4th Logic in Computer Science (1989).

[28] MENDELS ZONE, Demonstration at Internat. Conf. Fifth Generation Computer Systems (FGCS'92) (1992).

[29] F.A.Bernstein, N.Goodman, Concurrency Control in Distributed Database Systems, ACM Computing Surveys, Vol.13, No.2 (1981).

[30] A.Pnueli and R.Rosner, On the synthesis of an asynchronous reactive module,16th ICALP, LNCS 372 (1989).

[31] A.Pnueli and R.Rosner, On the synthesis of a reactive module, ACM POPL (1989).