NORTH-HOLLAND

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

Naoshi Uchihira and Shinichi Honiden

Systems & Software Engineering Laboratory, Research & Development Center, Toshiba Corporation 70, Yanagi-cho, Saiwai-ku, Kawasaki 210, JAPAN

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.

1. INTRODUCTION

1.1 Motivation

The difficulty of concurrent programming is mainly due to its nondeterministic behaviors. We classify nondeterminism into the following three 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, Figure 1 shows a simple Ada-like concurrent program "*Seat Booking*", where two processes read/write a shared memory "*seat*" to reserve one seat. This program has the three 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 the same result because l_1 (write in $status_1$) and m_1 (write in $status_2$)

Address correspondence to N. Uchihira, Systems and Software Engineering Laboratory, Research and Development Center, Toshiba Corporation 70, Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan.

*A running title is "Compositional Program Adjustment".

†*This is a revised version of the paper presented at the 28th Hawaii International Conference on Systems Sciences (1995).*

J. SYSTEMS SOFTWARE 1996; 33:207–221
© 1996 by Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010

0164-1212/96/$15.00
PII S0164-1212(96)00021-0

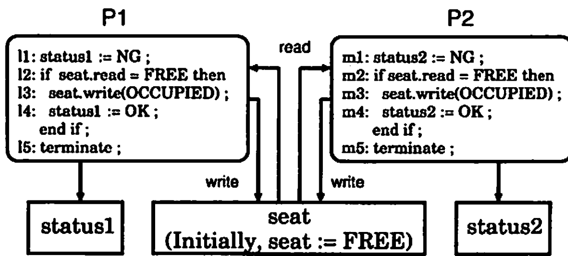

Figure 1. An example of a concurrent program.

are actions independent of each other. We call such a situation *persistent*.

- $\theta_4 = \mathbf{l}_1 \rightarrow \mathbf{m}_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK$, $seat = OCCUPIED$, $status_2 = NG$.
- $\theta_5 = \mathbf{m}_1 \rightarrow \mathbf{l}_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK$, $seat = OCCUPIED$, $status_2 = NG$.

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

We will show that this debugging process for harmful nondeterministic behaviors can be mechanically supported using the formal method (program synthesis techniques).

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 (i.e., its functional parts such as data transformation are correct) but may be imperfect in its timing (i.e., its synchronization parts have harmful nondeterministic behaviors). We call such a program a Functionally-Correct Temporally-Imperfect program (FCTI program).

A concurrent program is modeled with the finite state process (Kanellakis and Smolka, 1990), which can specify the finite state transition system with liveness conditions. It cannot only represent the transition systems in CCS (Milner, 1989), but also Büchi automata (Büchi, 1960). A target FCTI program is compositionally constructed from several finite state processes with the parallel composition operator "|" (e.g., $P = (P_{11} | P_{12}) | (P_{21} | P_{22})$ in Figure 2a).

Basic adjustment. Basic program 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 (Figure 2b). The arbiter partially serializes the FCTI program to remove harmful nondeterministic alternatives which do not satisfy given constraints. In the example of Figure 1, an arbiter can partially serialize the program so that the double booking is avoided (i.e., serialize it as $l_2 \rightarrow l_3 \rightarrow m_2$ or $m_2 \rightarrow m_3 \rightarrow l_2$). A concrete arbiter description (Figure 18), written in Ada for this example, will be explained in Section 6. We will show an algorithm to synthesize an arbiter process C_f automatically.

Input: An FCTI program P.
Input: Temporal logic constraints f.
Output: An arbiter C_f such that $P | C_f$ satisfies f.

Compositional adjustment. When a target program becomes large, the arbiter synthesis may cause an explosion in computing cost. 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 (Figure 2c).
$P' = (P_{11} | P_{12} | C_1) | (P_{21} | P_{22} | C_2) | C_0$.

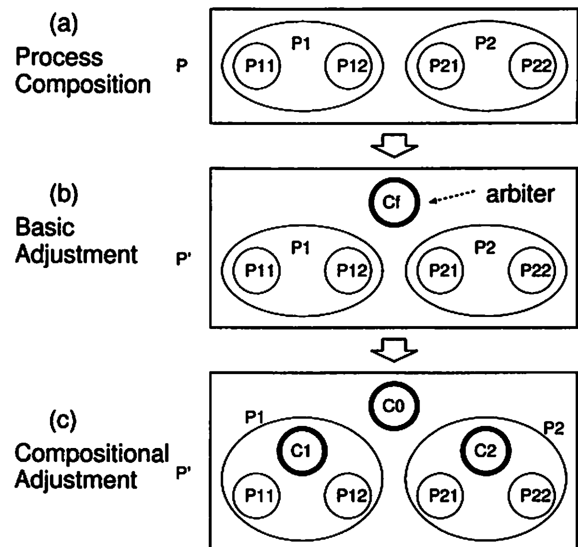

Figure 2. Process Composition, Basic Adjustment, and Compositional Adjustment.

In each composition step, the reduction of the finite state process, based on process equivalence theory, can ease the explosion in computing cost. 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.

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 (Figure 3).

1.3 Significance of the Paper

1. **Theoretical Aspect.** The traditional CCS framework (composition and equivalence) is not adequate for finite state processes with the liveness conditions (Büchi automata). Therefore, we introduce a new composition and equivalence for finite state processes that can preserve liveness properties. These techniques are essential to the basic and compositional adjustment, in which temporal logic constraints can be represented by the one of processes with liveness conditions.
2. **Practical Aspect.** We introduce the new concept of "compositional program adjustment" into concurrent programming, which is practical as compared with the program synthesis method from temporal logic specifications proposed in (Manna and Wolper, 1984). We have implemented the programming environment (MENDELS ZONE) adopting the program adjustment to show its effectiveness.

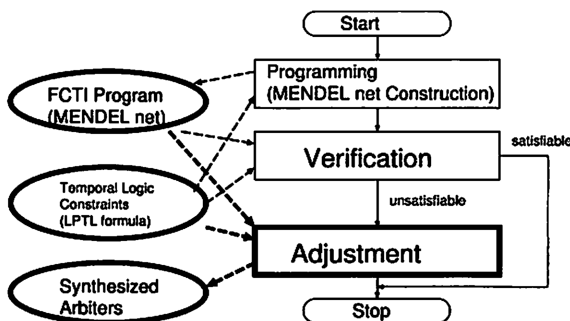

Figure 3. Verification and Adjustment in MENDELS ZONE.

1.4 Organization of the Article

The remainder of the article 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 shown and its compositional adjustment is explained in Section 4. Section 5 shows a simple and nontrivial example and an experimental result of compositional program adjustment. Section 6 takes program adjustment in standard programming languages into consideration, and is followed by the conclusion in Section 7.

2. FINITE STATE PROCESSES

The basic model for concurrent programs is the finite state process (Kanellakis and Smolka, 1990), 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 seventuple $P = (S, A, L, \delta, \pi, s_0, F)$, where

- S is a finite set of states;
- A is a finite set of actions;
- L is a finite set of synchronization labels;
- $\delta : S \times A \to S \cup \{\perp\}$ is a *deterministic* transition function ($\delta(s,t) = \perp$ means the action $t \in A$ is disabled in the state $s \in S$);
- $\pi : A \to (L \cup \{\tau\})$ 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 $\delta(s_0, t_1) = s_1$, $\delta(s_0, t_2) = s_2$, $\delta(s_1, t_2) = s_3$, $\delta(s_2, t_1) = s_3$, $\delta(s_3, t_3) = s_0$, $\pi(t_1) = a$, $\pi(t_2) = b$, $\pi(t_3) = \tau$. (Figure 4). ■

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

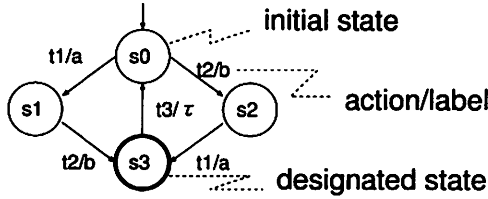

Figure 4. Finite state process.

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 $\theta[1]\theta[2]\ldots\theta[k]$ of θ, and $|\theta|$ 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 $\delta : S \times A^* \to S \cup \{\bot\}$, i.e., $\delta(s, \theta a) \stackrel{\text{def}}{=} \delta(\delta(s, \theta), a)$. Note, $\delta(s, \varepsilon) = s$. Because 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 $\pi : A^* \to (L \cup \{\tau\})^*$, i.e., $\pi(\theta) = \pi(\theta[1])\pi(\theta[2])\ldots\pi(\theta[|\theta|])$. 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 $R_P(s) \stackrel{\text{def}}{=} \{s' \in S \mid \exists\theta \in A^*.s' = \delta(s, \theta)\}$ and $R_P^+(s) \stackrel{\text{def}}{=} \{s' \in S \mid \exists\theta \in A^+.s' = \delta(s, \theta)\}$. Also, the set of all possible action sequences of P is defined as $L(P) \stackrel{\text{def}}{=} \{\theta \in A^* \mid \delta(s_0, \theta) \neq \bot\}$. Because 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*,

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

$L_{\omega}(P)$ 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 $L_{\omega}^{fair}(P) \stackrel{\text{def}}{=} \{\theta \mid \theta \in L_{\omega}(P)$ 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: $L(P)/A' \stackrel{\text{def}}{=} \{\theta' \mid \exists\theta \in L(P).\forall i.(\theta'[i] = \varepsilon$ 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 states 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 $k \geq 1$. $L_{sat}(P) \subset A^{\omega}$ is defined as a set of all satisfiable behaviors[1] on P. ■

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*. If P is complete, P has no unsatisfiable states. A behavior reaching an unsatisfiable state is called an *inevitably unsatisfiable behavior*.

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

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 CCS's weak bisimulation equivalence (Milner, 1989). In this article, it is used to reduce an FSP to a smaller and equivalent FSP 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 $\forall n > 0.\exists\theta \in A^*.(|\theta| = n$, $\hat{\pi}(\theta) = \varepsilon$ *and* $\delta(s, \theta) \neq \bot)$. ■

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 $\forall s_1 \in S_1.\forall s_2 \in S_2.(s_1, s_2) \in R \Leftrightarrow$

- $s_1 \in F_1$ *iff* $s_2 \in F_2$;
- $s_1\uparrow$ *iff* $s_2\uparrow$;

[1] A satisfiable behavior corresponds to an accepting run of the Büchi automaton.

J. SYSTEMS SOFTWARE 1996, 33:207-221

- $\forall t_1 \in A_1.\forall s_1' \in S_1.(if\ \ s_1' = \delta_1(s_1, t_1)$ 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 that 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 $|S_r| \leq |S|$. ■

The smallest $red(P)$ is constructed effectively by the relational coarsest partitioning algorithm (Page and Tarjan, 1987; Kanellakis and Smolka, 1990) 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 $P = P_1 | P_2$ is defined as follows.

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

- $\delta : (S_1 \times S_2 \times \{0,1\}^2) \times (A_1 \cup \{idle\}) \times (A_2 \cup \{idle\}) \to (S_1 \times S_2 \times \{0,1\}^2) \cup \{\perp\}$ *such that*

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

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

$$\begin{cases} \pi((a_1, a_2)) = l & \text{if } a_i \in A_i, \\ & \pi_1(a_1) = \pi_2(a_2) = l \\ \pi((a_1, idle)) = \pi_1(a_1) & \text{if } a_1 \in A_1, \\ \pi((idle, a_2)) = \pi_2(a_2) & \text{if } a_2 \in A_2, \\ \pi((a_1, a_2)) = \tau & \text{otherwise} \end{cases}$$

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

We remark that processes are synchronized at actions *with the same labels* in the above process composition. This composition is similar to composition in CCS (Milner, 1989) 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 the same labels.

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

$$P' = (S, A, L', \delta, \pi', s_0, F), \text{ where } \pi'(a) = f(\pi(a))$$
$$\text{if } \pi(a) \neq \tau; \text{ otherwise, } \pi'(a) = \tau. \quad ■$$

Example 2. (Composition and Relabeling) Figure 5 shows an example process composition and relabeling: $P_1[f_1] | P_2[f_2]$ where 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* $i = 1, 2$). ■

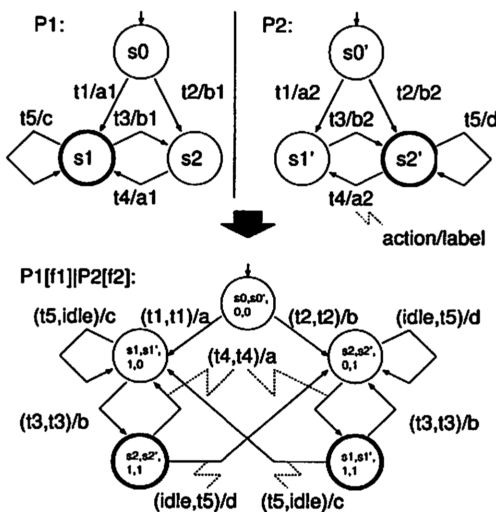

Figure 5. Composition and Relabeling.

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

Lemma 3. (Reflectivity) Let P_1 and P_2 be FSPs. If $P = P_1 | P_2$, then $L_{sat}(P) \downarrow left \subset L_{sat}(P_1)$ and $L_{sat}(P) \downarrow right \subset L_{sat}(P_2)$. ■

Lemma 4. (Substitutivity) $\pi\tau\omega$-bisimulation equivalence is preserved by composition and relabeling; that is, if $P \approx_{\pi\tau\omega} Q$, then $\forall R.(P | R \approx_{\pi\tau\omega} Q | R)$, 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 FCTI process P and a temporal logic constraint f, $P | P_f$ is a composed process in which P's behaviors against f are disabled by P_f (i.e., safety properties are satisfied). However, $P | P_f$ is not necessarily complete (i.e., liveness properties may not be satisfied). Program adjustment means to make $P | P_f$ complete by adding an arbiter process C (i.e., the adjusted program $= P | P_f | C$).

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 of actions.

Definition 10. (LPTL)

Syntax: Linear time propositional temporal logic (LPTL) formulas are built from atomic propositions *Prop*, Boolean connectives ($\wedge$, $\neg$), and temporal operators (O("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 temporal operators intuitively have the following meanings.

- $\bigcirc f$ (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 (Manna and Wolper, 1984). ■

We use $\diamond f$ ("eventually f") as an abbreviation for *true* Uf and $\square f$ ("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[2], 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 $L_{sat}(P_f)$ is exactly the set of behaviors whose label sequences satisfy f[3].

Proof. It is a restriction of a general theorem (Wolper et al., 1983). ■

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

(1) $\square\diamond(a_1 \vee a_2)$: Either a_1 or a_2 must infinitely often occur.
(2) $\square(a_1 \Rightarrow \bigcirc\square(\neg a_2))$: Whenever a_1 occurs, then a_2 must never occur.

FSPs which are generated from (1) and (2) are shown in Figure 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, and identify a label and an action ($A = L$ and $\pi(a) = a$). In this case, some formulas, such as $\diamond\square a$, which are translated to nondeterministic FSP, become unavailable. These formulas are suitable for verification, but not for adjustment (synthesis) because the arbiter cannot look ahead to future behaviors, as indicated by Pnueli and Rosner (1989).

[2] Other propositional modal logics such as μ-calculus are also applicable to our theories.
[3] A label sequence of a satisfiable behavior of P_f corresponds to a model of an LPTL formula.

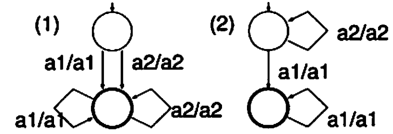

Figure 6. FSPs P_f of Temporal Logic Constraints.

3.2 Basic Adjustment

When temporal logic constraints f can be translated to an FSP P_f, we have to show how to make an FSP $P = P_f | P_0$ *complete* for the target FCTI program P_0 by adding an arbiter process C. In other words, basic adjustment is defined as an arbiter synthesis for $P = P_f | P_0$ (Figure 7).

In the following explanation, we assume that the target FSP P has already composed with P_f (i.e., $P = P_f | \ldots$) and do not mention P_f explicitly.

Problem 1. (Basic Adjustment)

Input An FSP $P = (S, A, L, \delta, \pi, s_0, F)$.

Output A maximally permissive FSP $C = (S_c, A_c, L_c, \delta_c, \pi_c, s_{0c}, F_c)$ such that $P|C$ is complete. "C is maximally permissive" means that for every C' if $P|C'$ is complete then $L(P|C') \subset L(P|C)$. ■

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 $C_\tau(s, a) \overset{\text{def}}{=} \{s' | \exists\theta.(s' = \delta(s, \theta), \hat{\pi}(\theta) = a)\}$

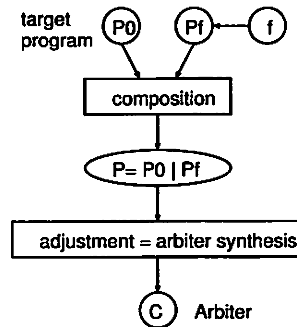

Figure 7. Basic Adjustment.

for $\forall s \in S'$ and $\forall a \in L \cup \{\varepsilon\}$, $C_\tau(S_{sub}, a) \stackrel{\text{def}}{=} \bigcup_{s \in S_{sub}} C_\tau(s,a)$ for $\forall S_{sub} \subset S'$ and $\forall a \in L \cup \{\varepsilon\}$, then C' is defined as $C' = (S'_c, A'_c, L, \delta'_c, \pi'_c, C_\tau(s'_0, \varepsilon), S'_c)$ 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* $C_\tau(s', a) \cap S_u = \varnothing$,
- $\delta'_c(s', t_a) = \bot$ *if* $C_\tau(s', a) \cap S_u \neq \varnothing$[4],
- $\delta'_c(s', t_{s'}) = s'$,

and $\pi'_c(t_a) = a$ and $\pi'_c(t_{s'}) = \tau$ for $\forall a \in L$, $\forall s' \in S'_c$.

(Step 3) $P' := P' | C'$, 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 $P|C$ 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. Because the transition function of C' is deterministic respecting its labels, C' restrains no satisfiable behavior of P. Therefore, $P|C$ is complete and C is maximally permissive. ■

Corollary 1.
$L^{fair}_\omega(P|C) \downarrow left \subset L_{sat}(P|C) \downarrow left \subset L_{sat}(P)$
(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. We remark that an arbiter is effective in case $L^{fair}_\omega(P) \not\subset L_{sat}(P)$ (i.e., P has some harmful nondeterministic behaviors). Finally, we have the following result.

Corollary 2. The adjusted process $P_f | P_0 | C$ satisfies the temporal logic constraint f.

(Proof). It is a special case of Corollary 1, that is, $L^{fair}_\omega(P_f|(P_0|C)) \downarrow left \subset L_{sat}(P_f)$. ■

[4] "$\delta'_c(s', t_a) = \bot$ if $C_\tau(s', a) \cap S_u \neq \varnothing$" means elimination (disablement) of all behaviors that cannot be distinguished from inevitably unsatisfiable behaviors by a label observer.

Example 4. (A single arbiter synthesis) Figure 8 shows a simple single arbiter synthesis. In the target process P, only $\theta = t_3 t_6 t_7$ is an inevitably unsatisfiable behavior. Because $\{t_3 t_6 t_7, t_3 t_4\}$ is a set of behaviors that 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.

3.3 Compositional Adjustment

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

1. The synthesis results in an explosion in computing cost;
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 (minimization) of an FSP based on $\pi\tau\omega$-bisimulation can ease the explosion in computing cost 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, $C|P_1$ is complete iff $C|P_2$ is complete. ■

Corollary 3. If C is an arbiter of *red*(P), then C is also an arbiter of P. ■

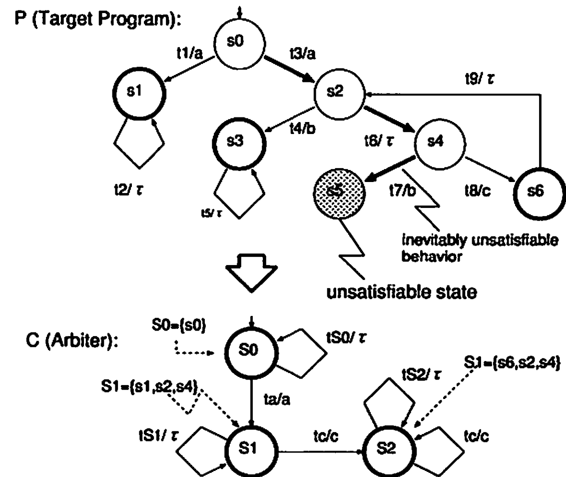

Figure 8. Single Arbiter Synthesis.

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

- **Target Program (Figure 2c)**
  $(P_{11}[h_{11}] \mid P_{12}[h_{12}])[h_1] \mid (P_{21}[h_{21}] \mid P_{22}[h_{22}])[h_2]$
  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 (Figure 9).

(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 $P_1 \stackrel{\text{def}}{=} (C_1 \mid P_{11}[h_{11}] \mid P_{12}[h_{12}] \mid P_{f_1})[h_1]$ and $P_2 \stackrel{\text{def}}{=} (C_2 \mid P_{21}[h_{21}] \mid P_{22}[h_{22}] \mid P_{f_2})[h_2]$.

(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 3 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. It is possible to directly synthesize 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 already been proposed and shown its effectiveness in compositional verification by Clarke et al. (1989). However, the reduction *preserving liveness properties* by $\pi\tau\omega$-bisimulation is our original work.

4. MENDELS ZONE

4.1 Overview

MENDELS ZONE is a programming environment for concurrent programs. It was developed over eight years by a team comprising less than 10 people as a part of the Fifth Generation Computer System (FGCS) project. The target concurrent programming language, MENDEL (Uchihira et al., 1992), is based on a high-level Petri net. It is translated into the concurrent logic programming language KL1 (Ueda and Chikayama, 1990) and executed on the parallel machine Multi-PSI (Taki, 1989). MENDEL is regarded as a macro language of KL1.

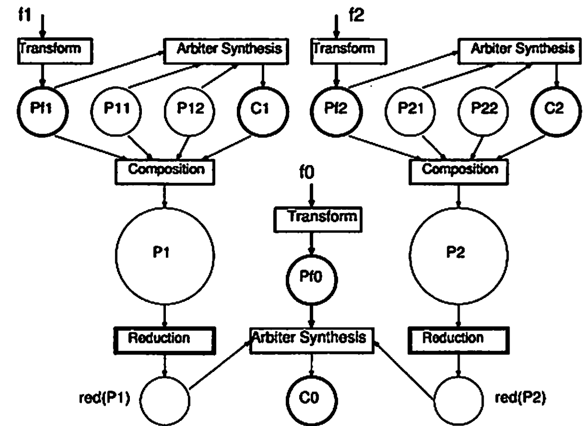

Figure 9. Compositional Arbiter Synthesis.

MENDELS ZONE supports (1) construction of MENDEL atomic processes, (2) graphical process interconnection (Uchihira et al., 1992), (3) compositional adjustment of interconnected MENDEL processes based on theories described in Section 3, and (4) performance design (Honiden et al., 1994).

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 Figure 10, 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. MENDEL net is extended from a Petri net in the following aspects.

- Modularity is introduced. A module of MENDEL net represents a process.
- Synchronous (handshake) communication between processes is introduced, in addition to asynchronous (dataflow) communication.
- Each transition can have an additional enable condition that must be satisfied when the transition fires, and an additional action, which is executed when it fires. Both are written by KL1[5].

[5]KL1 codes attached to transitions are ignored in the adjustment.

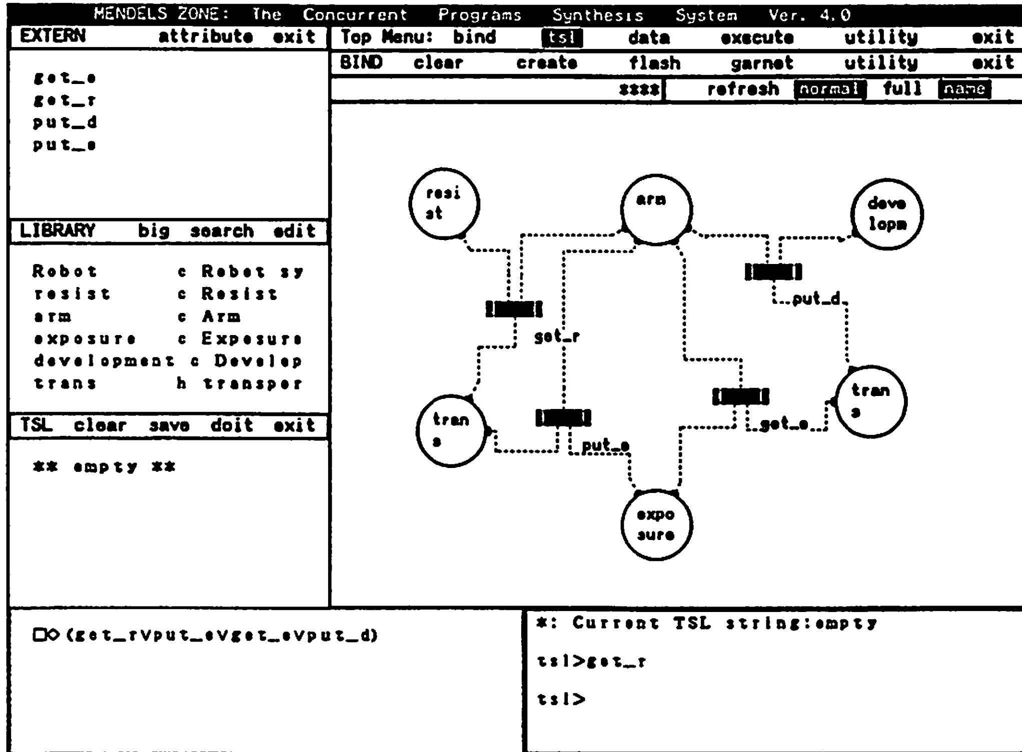

Figure 10. MENDELS ZONE.

MENDEL net is graphically represented like a Petri net (Figure 11). The basic conventions are as follows.

- Each "place" is represented by a circle.
- Each "transition" is represented by a square.

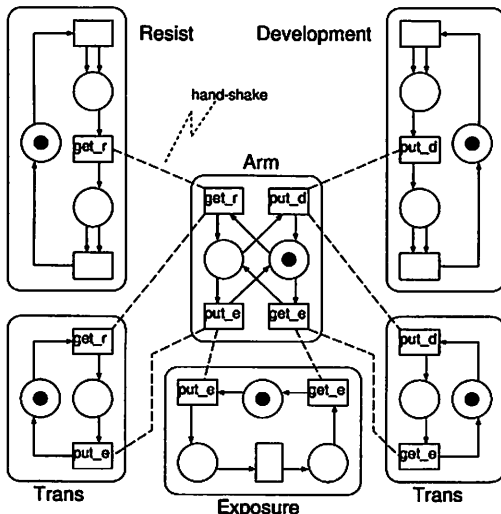

Figure 11. MENDEL net.

- Each process is represented by enclosing places and transitions belonging to the process with a line.
- A "synchronous (handshake) communication" is represented by a dashed 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 with only synchronous communications between processes, which can be translated into FSPs.

4.3 Programming in MENDELS ZONE

(Phase 1) **MENDEL Net Construction.** A programmer can construct a MENDEL net using the graphic editor and the program library as follows.

- (Step 1) Construct atomic MENDEL processes basically by software reuse (Uchihira et al., 1987).
- (Step 2) Interconnect MENDEL processes with communication links using the graphic editor to

J. SYSTEMS SOFTWARE 1996, 33:207-221

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.

(Phase 2) **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 (Figure 3) in MENDELS ZONE is as follows.

1. The programmer can give an LPTL formula for a MENDEL net of some compound process.
2. MENDELS ZONE checks whether a MENDEL net satisfies a given LPTL formula by the model checking method for LPTL (Vardi and Wolper, 1986).
3. When it does not satisfy the LPTL formula, the adjustment method is invoked.

(Phase 3) **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 by an execution monitor.

5. EXAMPLE AND EXPERIMENTAL RESULT

5.1 Example: 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 (Figure 12).

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* × 2) that control corresponding machines. *Trans* represents board transportation. Each process is displayed as a MENDEL net, shown in Figure 11. With no arbiter, this system is FCTI because it falls into deadlock when an action label sequence of Arm

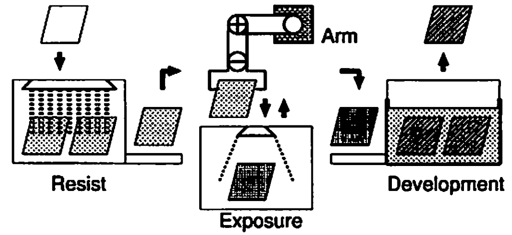

Figure 12. Machine for processing printed circuit boards.

"$get_r \rightarrow put_e \rightarrow get_r$" occurs. We give the following temporal logic constraints

$$f = \Box\Diamond(get_r \vee put_e \vee get_e \vee put_d),$$

which means Arm never fails into deadlock. An arbiter C is synthesized as follows: first, FSPs representing six 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 (Figure 13) representing temporal logic constraints f is generated. The target process P (Figure 14) is composed from these FSPs (including P_f). Finally, the arbiter C shown in Figure 15 is synthesized from P, according to Algorithm 1. We can see that the adjusted program "$C \| P_f \| Resist[f_r] \| Exposure[f_e] \| Development[f_d] \| Arm[f_a] \| Trans[f_{t_1}] \| Trans[f_{t_2}]$" satisfies the above constraints. Figure 16 shows the adjusted program represented by MENDEL net. You can see the target MENDEL net in Figure 11 is adjusted by introducing the arbiter C in Figure 16.

5.2 Experimental Result

We will show how well the compositional method works when it is applied to a middle-scale manufacturing machine control software. This machine is controlled by a concurrent (multi-task) program which consists of 16 element processes (tasks). Table 1 shows the sizes of element processes. The state numbers of each element process may seem small. It

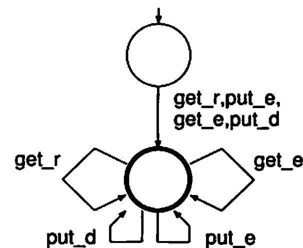

Figure 13. FSP P_f for LPTL formula f.

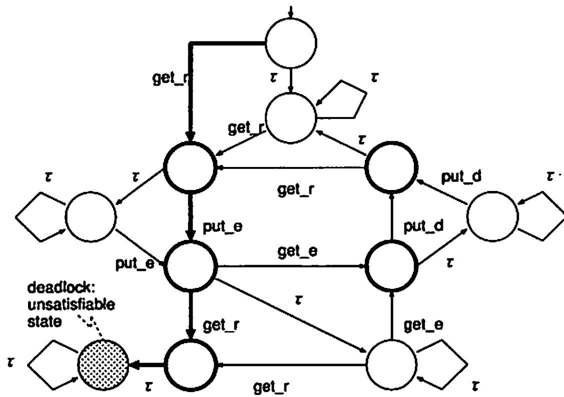

Figure 14. Target Process P (displaying only labels).

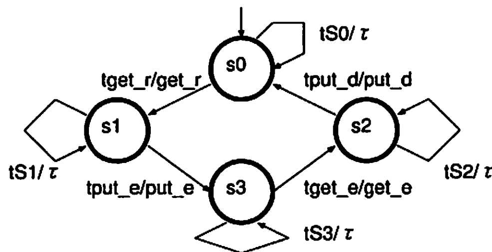

Figure 15. Synthesized Arbiter C.

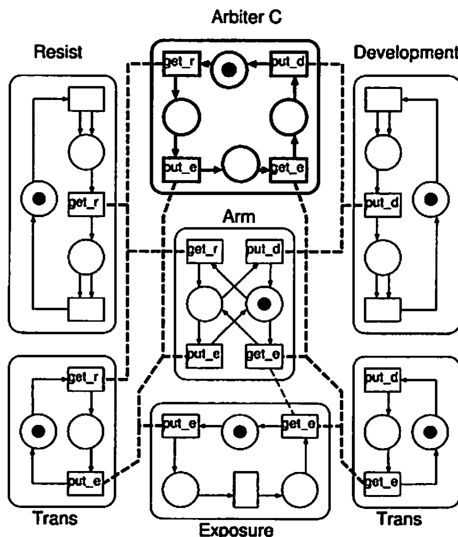

Figure 16. Adjusted Program.

Table 1. Middle-scale Machine Control Software

Element FSP	Number of States
(P_1) Distribution Arm	6
(P_2) Testing Equipment	3
(P_3) 1st Manufacturing Equipment	5
(P_4) 2nd Manufacturing Equipment × 2	3
(P_5) 3rd Manufacturing Equipment	5
(P_6) Set-up Arm × 2	6
(P_7) Extracting Arm × 2	6
(P_8) 1st door × 2	2
(P_9) 2nd door × 2	3
(P_{10}) Conveyer × 2	3

is attributable to the fact that only synchronization parts of systems are modeled by FSPs.

For these target processes, we give temporal logic constraints by f; prohibition of illegal behaviors of arms and deadlock-freedom for two symmetric process groups $(P_4, P_6, P_7, P_8, P_9, P_{10})$. Two arbiters were synthesized after the compositional adjustment procedure. Figure 17 shows their whole structure (communication and synchronization among processes).

The compositional adjustment procedure to synthesize two arbiters C_{f1} and C_{f2} is shown as follows[6].

1. $P_{c1} = red(P_4 | P_6 | P_7 | P_8 | P_9 | P_{10})$ (max_size = 48 states)
2. $P_{c2} = red(P_1 | P_2 | P_3 | P_5)$ (max_size = 85 states)
3. $P_{c3} = red(P_{c1} | P_{c2})$ (max_size = 77 states)
4. The first arbiter C_{f1} is synthesized from f and P_{c3} (max_size = 385 states)
5. $P_{c4} = red(P_{c3} | C_{f1})$ (max_size = 88 states)
6. $P_{c5} = red(P_{c4} | P_{c1})$ (max_size = 239 states)
7. The second arbiter C_{f2} is synthesized from f and P_{c5} (max_size = 112 states)

Here, the "max_size" means the maximal number of states which are temporally created during process composition and reduction procedure at each step, and the worst case is *max_size* = 385. Without the compositional method (i.e., by the basic adjustment), the naive process composition of 16 processes would generate a far larger number of states since the *max_size* is increasing monotonously without reduction. Table 2 shows the maximum number of states in two cases (basic adjustment and compositional adjustment). It indicates that the compositional method can reduce the maximum size to about 1/150 of the basic adjustment.

In this example, only synchronization parts of the system are modeled by FSPs. If they have a lot of

[6] Relabeling functions are omitted for simplicity.

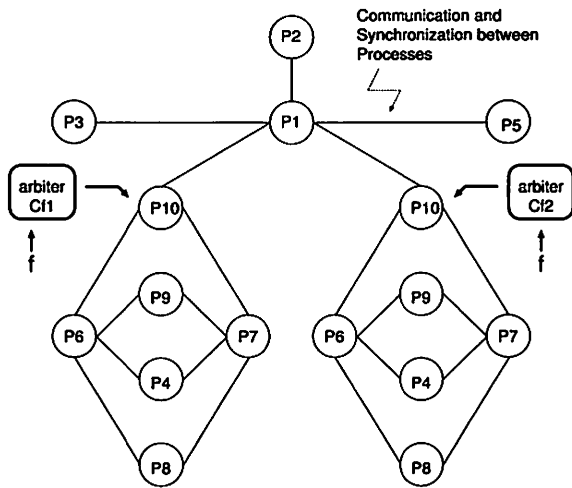

Figure 17. Adjusted middle-scale machine control software.

actions unrelated to synchronization, which are regarded as τ actions, the process reduction would be more effective.

6. PROGRAM ADJUSTMENT IN STANDARD PROGRAMMING LANGUAGES

This section briefly considers program adjustment in standard programming languages, instead of MENDEL net. Program adjustment is applicable to concurrent programming languages which have a synchronous (i.e., handshake) communication mechanism, like Ada and Occam. For example, Figure 18 shows a program adjustment example for the Ada program used in the motivation section (Figure 1). Two FSPs P_1 and P_2 are derived from the original program, then an arbiter is synthesized by the basic adjustment procedure, and finally an adjusted Ada program[7] is derived from FSPs and the arbiter. As you can see in Figure 18, the arbiter controls the target programs using a *rendezvous* mechanism of Ada to remove harmful nondeterministic behaviors (i.e., θ_3) mentioned in Section 1.1.

When applying the program adjustment to Ada, we require the following two converters.

- **Ada → FSP converter**: The Ada program code is divided into basic blocks. Each basic block is assigned to one state of a generated FSP. Control flows between basic blocks are represented as edges between these states. Synchronous communication commands are also represented as edges with synchronization labels. Furthermore, the user can put arbitrary labels on edges which are used to specify temporal logic constraints.
- **FSP → Ada converter**: A synthesized arbiter represented by an FSP is converted into an Ada task which implements state transitions using *loop* and *select* constructs. Synchronization labels in the arbiter are converted into *accept* commands, and synchronization labels in the target processes are converted into *entry call* commands.

[7]Some trivial declarations are omitted.

Table 2. Middle-scale Machine Control Software (Effect of Process Reduction)

Adjustment Type	Maximum Temporary Size of States
Basic Adjustment	61096
Compositional Adjustment	385

7. CONCLUSIONS AND RELATED WORKS

We have introduced the concept of "program adjustment" into concurrent programming. For compositional adjustment, we have also introduced a new composition and equivalence for finite state processes which can preserve liveness properties.

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 (Manna and Wolper, 1984; Emerson and Clarke, 1982). The reasons are as follows.

- It is not 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 only timing constraints, such as deadlock-free and starvation-free constraints, as compared with specifying a whole specification.
- Computing cost of program adjustment is lower than that of program synthesis.

We also remark that our method is suited for reactive systems which have uncontrollable and unobservable elements in their environments because they can be modeled by τ actions in FSP.

We have some experience of state-transition-based software construction, using compositional adjust-

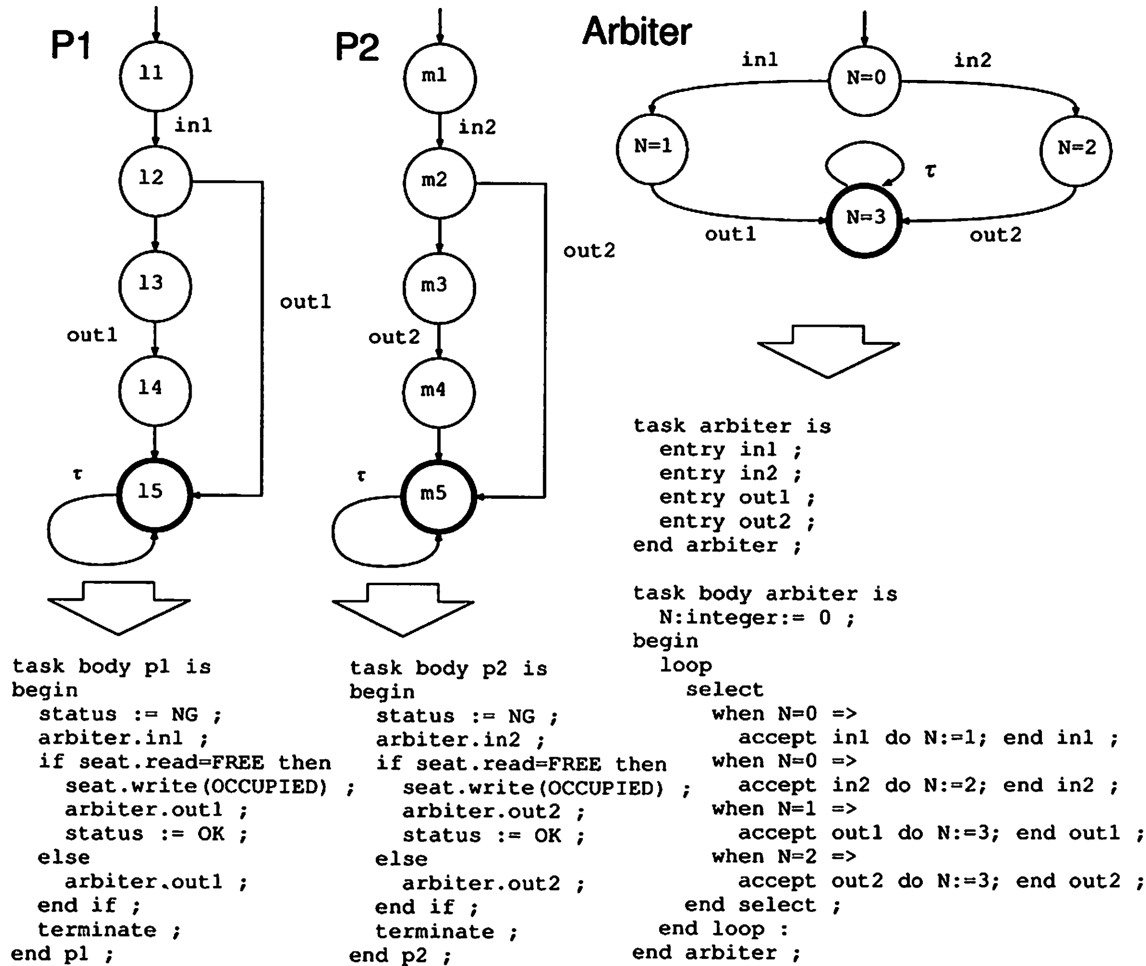

Figure 18. Program Adjustment in Ada.

ment in MENDELS ZONE. For example, we have constructed a control software for a power plant (about 4,300 steps) which consists of 6 processes and evaluated MENDELS ZONE (MENDELS ZONE, 1992). In this case, timing bugs between sensor inputs and data change detection are adjusted.

In our previous works (Uchihira et al., 1987; Uchihira et al., 1990; Uchihira and Honiden, 1990) we proposed program synthesis methods based on temporal logic. However, these methods generated a global state transition graph based on the assumption that all process actions are observable (not internal) and controllable, that is, not reactive. This assumption is restrictive because most practical systems have some reactive features. Moreover, the global state transition graph often becomes huge, and its generation is costly because it cannot be done compositionally. In this article, we introduce a CCS-like compositional framework to deal with unobservable and uncontrollable actions and to achieve compositional adjustment utilizing process reduction.

Abadi, Lamport, and Wolper (Abadi et al., 1989) 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 (Ramadge and Wonham, 1989). However, while these works mainly consider safety properties, they showed no compositional synthesis methods satisfying liveness constraints. The concurrency control of database transactions (Bernstein and Goodman, 1981) is intimately 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, and we would like to thank the members of that organization. We are also grateful to Sadakazu Watanabe and Kazuo Matsumura of the Systems & Software Engineering Laboratory, Toshiba Corporation, for providing support throughout this work.

REFERENCES

Abadi, M., Lamport, L., and Wolper, P., Realizable and Unrealizable Specifications of Reactive Systems, 16th ICALP (1989).

Bernstein, P. A., and Goodman, N., Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, 13 (2), 185–221 (1981).

Büchi, J. R., A Decision Method in Restricted Second Order Arithmetic, in *Proc. Internat. Congr. Logic, Method. and Philos. Sci.* (1960), Stanford University Press, 1962.

Clarke, E. M., Long, E. E., and McMillan, K. L., Compositional Model Checking, in *Proc. 4th Symposium on Logic in Computer Science*, pp. 353–362 (1989).

Emerson, E. A., and Clarke, E. M., Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons, *Science of Computer Programming*, Vol. 2, 241–266 (1982).

Honiden, S., Nishimura, K., Uchihira, N., and Itoh, K., An Application of Artificial Intelligence to Object-Oriented Performance Design for Real-Time Systems, *IEEE Trans. Software Engineering*, 20 (11), 849–867 (1994).

Kanellakis, P. C., and Smolka, S. A., CCS Expressions, Finite State Processes and Three Problems of Equivalence, *Information and Computation*, 86, 43–68 (1990).

Manna, Z., and Wolper, P., Synthesis of Communicating Processes from Temporal Logic Specification, *ACM Trans. Program. Lang. & Syst.* 6 (1), 68–93 (1984).

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

Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.

Paige, R., and Tarjan, R. E., Three Partition Refinement Algorithms, *SIAM J. Comput.* 16 (6), 973–989 (1987).

Pnueli, A., and Rosner, R., On the Synthesis of a Reactive Module, *16th ACM Symp. on Principles of Programming Languages*, 179–190 (1989).

Ramadge, P. J., and Wonham, W. M., The Control of Discrete Event Systems, *Proc. IEEE*, 77 (1), 81–98 (1989).

Taki, K., The FGCS Computing Architecture, in *Information Processing 89, Proc. IFIP 11th World Computer Congress*, 1989, pp. 627–632.

Uchihira, N., Kasuya, T., Matsumoto, K., and Honiden, S., Concurrent Program Synthesis with Reusable Components Using Temporal Logic, in *Proc. 11th Internat. Computer Software & Applications Conference* (COMPSAC), 1987, pp. 455–464.

Uchihira, N., Kawata, H., Matsumoto, K., Ito, M., and Honiden, S., Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse, in *Proc. 23rd Hawaii International Conference on Systems Science* (HICSS), Vol. 2, 1990, pp. 64–73.

Uchihira, N. and Honiden, S., Verification and Synthesis of Concurrent Programs using Petri Nets and Temporal Logic, *Trans. IEICE*, E73 (12), 1001–2010 (1990).

Uchihira, N., Arami, M., and Honiden, S., A Petri-Net-Based Programming Environment and Its Design Methodology for Cooperating Discrete Event Systems, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E75-A (10), 1335–1347 (1992).

Ueda, K., and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine, *Comput. J.*, 33 (6), 494–500 (1990).

Wolper, P., Vardi, M. Y., and Sistla, A. P., Reasoning about Infinite Computation Paths, in *Proc. IEEE 24th Symp. on Foundations of Computer Science*, 185–194 (1983).

Vardi, M. Y., and Wolper, P., An Automata-Theoretic Approach to Automatic Program Verification, in *Proc. 1st Symp. on Logic in Computer Science* 332–344 (1986).

APPENDIX A. THE NOTATIONS

Several notations which are frequently used in this paper are summarized as follows.

- $P_1 | P_2$ stands for parallel composition of P_1 and P_2.
- $P[f]$ stands for relabeling of P by a relabeling function f.
- $red(P)$ stands for process reduction of P.
- $P_1 \approx_{\pi\tau\omega} P_2$ means that P_1 and P_2 are $\pi\tau\omega$-bisimulation equivalent.
- $L_{sat}(P)$ stands for a set of all satisfiable behaviors in P.
- $L_{\omega}^{fair}(P)$ stands for a set of all infinite behavior in P under the fairness condition.
- $L(P_1 | P_2)\downarrow left$ means a left projection of $L(P_1 | P_2)$.
- $\Diamond f$ means a formula f will eventually become true.
- $\Box f$ means a formula f as always true.