International Journal of Parallel Programming, Vol. 28, No. 2, 2000

Scenario-Based Hypersequential Programming

Naoshi Uchihira,[1] Hideji Kawata,[1] and Fumitaka Tamura[1]

Received March 21, 1998; revised September 16, 1998

Hypersequential programming is a new paradigm of concurrent programming. The original concurrent program is first serialized, then the sequential version is tested and debugged, and finally the target concurrent program is synthesized by parallelizing the debugged sequential version. In hypersequential programming, testing and debugging are performed on the sequential version of the program and the correctness is preserved in the subsequent parallelization process. Therefore, it offers both higher productivity and enhanced reliability. This paper describes a practical approach to hypersequential programming using the execution history called *scenario*. It also formalizes the parallelization process using a new equivalence relation called *scenario graph equivalence*, and gives the parallelization algorithm.

KEY WORDS: Concurrent programming; testing; debugging; Petri net; communicating transition systems; parallelization; hypersequential programming; scenario; timing tranquilizer.

1. INTRODUCTION

A rapidly growing trend towards parallel and distributed computer systems has increased demand for programmers writing concurrent application programs. In particular, Java has made multithreaded concurrent programming remarkably popular. However, concurrent programs are in general more difficult to develop than sequential programs.(1) In particular, testing and debugging are major bottlenecks in concurrent programming.(2) *Hypersequential programming*(3) is a new programming paradigm to

[1] Corporate Research and Development Center, Toshiba Corporation, E-mail: naoshi.uchihira@{kawata@ssel.,tamu@ssel.}toshiba.co.jp.

0885-7458/00/0400-0155$18.00/0

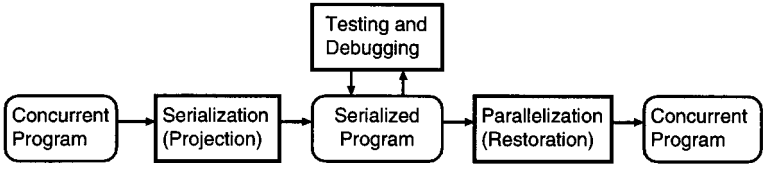


Fig. 1. Concept of hypersequential programming.

provide a solution to these bottlenecks. Figure 1 illustrates the concept of hypersequential programming. The original concurrent program is first serialized. Then testing and debugging are performed on the sequential version. Finally the reliable version of the concurrent program is reconstructed through parallelization of the debugged sequential version. Hypersequential programming makes concurrent programs as easy to test and debug as sequential programs since testing and debugging are performed on the serialized version. The correctness of the debugged sequential version is preserved in the parallelization process. Therefore, the generated concurrent program is as reliable as the sequential version.

Hypersequential programming can be implemented in a variety of ways. Uchihira *et al.* have presented an implementation based on petri-net-rewriting.[3] In this paper, another practical approach, called *scenario-based hypersequential programming*, is proposed (Fig. 2). A *scenario* is a sequential execution history that the programmer tests. The programmers can construct a set of scenarios (represented by *scenario graph*) as if they were performing conventional sequential program testing. The parallelization algorithm presented in this paper uses such scenarios to automatically reconstruct the final concurrent program.

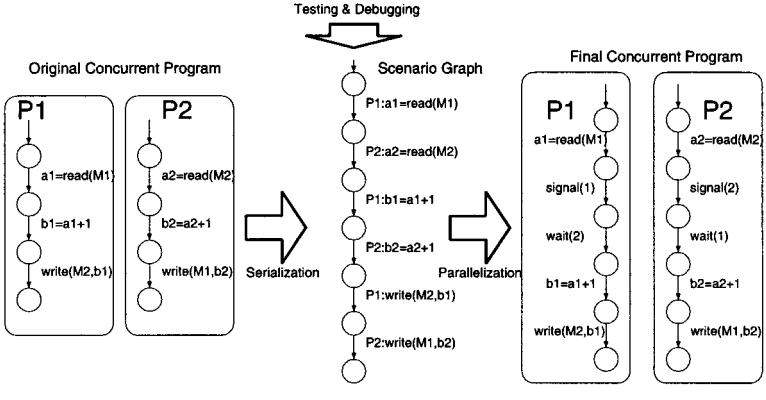


Fig. 2. Simple example of scenario-based hypersequential programming.

The rest of the paper is organized as follows: Section 2 shows an overview of scenario-based hypersequential programming. Section 3 formalizes the approach and gives a parallelization algorithm in detail. Section 4 uses an example to present how scenario-based hypersequential programming works. Sections 5 and 6 give related works and a conclusion, respectively.

2. SCENARIO-BASED HYPERSEQUENTIAL PROGRAMMING

In scenario-based hypersequential programming, programmers develop a concurrent program in the following four steps:

Step 1. Modeling the target system and coding the initial concurrent program. Model the target system using concurrency and nondeterminacy. Write a program in a concurrent programming language (e.g., Ada, Java, C with a multi-tasking library), exploiting such concurrency and nondeterminacy.

Step 2. Testing the program under a set of scenarios and constructing a scenario graph.

Step 2.1. Execute the concurrent program sequentially (step-by-step) and store its execution history. The resulting execution history is called a *scenario*. In this step, a conventional-debugger-like graphical user interface (GUI) is helpful. It allows the programmer to interactively specify which process (or statement) is executed next (Fig. 3). In this figure, executable processes (or statements) are displayed in a menu at each step, then the programmer selects one of them, and the selected process (or statement) is executed by one step. A sequence (path) of selected statements represents one scenario in the scenario graph browser.

Step 2.2. Using a conventional testing and debugging scheme for sequential programs, make sure each scenario satisfies the programmer's intention. If bugs are detected, go back to Step 1 and modify the original source code. Otherwise, go back to Step 2.1 for more scenarios until all test cases that the programmer intends are inspected.

Step 2.3. After all the intended scenarios are obtained, a global state transition system, called a *scenario graph*, is constructed. In the scenario graph, a node is a global program state. An edge is a program statement which caused the transition between the global states. A scenario is represented by a path from the initial node. Such a path represents a feasible intended behavior of the concurrent program. Note that the scenario graph does not contain untested and unexpected nondeterministic behaviors which often cause serious bugs.

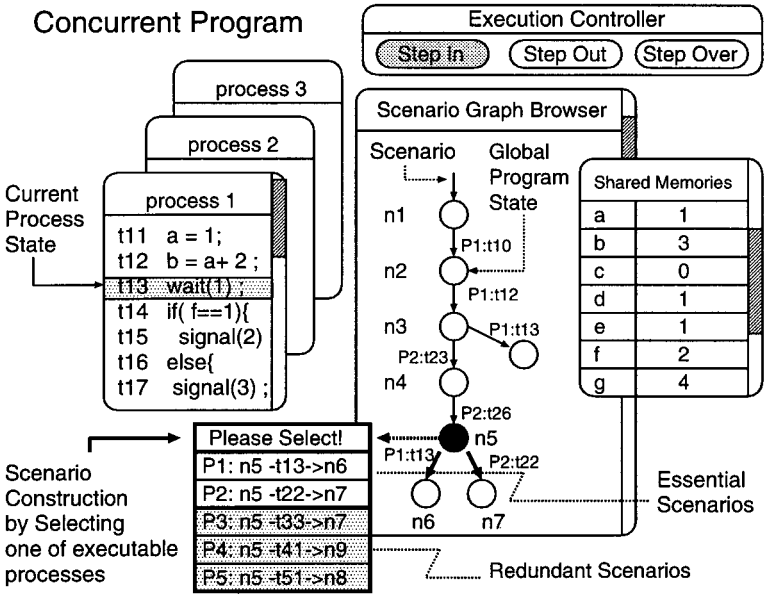


Fig. 3. Scenario selection and testing tool.

Step 3. Parallelization of the scenario graph involves the following three steps: insertion of pseudo-synchronization operations (adding nodes and edges to the graph), dividing the graph to processes (constructing the local scenario graphs), and removing redundant pseudo-synchronization from the local scenario graphs (Fig. 4).

Step 3.1. Insert pseudo-synchronization actions sync(ID) into the scenario graph. This will be used for a global synchronization between processes in the next step. For the sake of simplicity, synchronization actions are inserted, in this discussion, after every statement and before every branch statement (Fig. 4b).

Step 3.2. Divide the scenario graph to the set of local scenario graphs by projection. The projection assigns a statement-action (a t-edge) to the process it originally belongs to, and assigns empty actions (ε actions) to other processes. It also assigns all synchronization-actions (s-edges) to all processes. Nodes of the local scenario graph are first created by corresponding to nodes of the original scenario graph, then every two nodes connected with an ε action are shrunk into one node. For example, the local scenario graph of P_1 in Fig. 4c consists of P_1's own statements (t-edges: t_{1_2}, t_{12}, t_{13}, t_{14}, t_{15}) and pseudo-synchronization actions (s-edges:

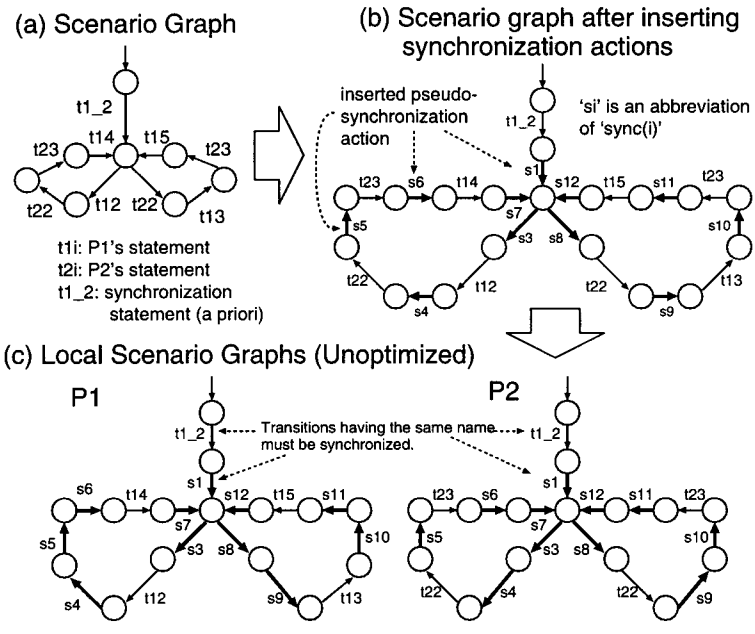


Fig. 4. Parallelization from scenario graph $SG_{\mathscr{C}}$.

$s_1, ..., s_{12}$). It does not contain P_2's local statements (t_{22}, t_{23}) because these are shrunk as ε actions. The unoptimized local scenario graphs represent graphical versions of concurrent processes which are synchronized with each other step-by-step, using explicit pseudo-synchronization actions. Note that the concurrent program composed of these unoptimized local scenario graphs faithfully reproduces one of the scenarios (intended behaviors) specified by the programmer. The inserted pseudo-synchronization actions guarantee this behavior.

Step 3.3. Optimize each local scenario graph by removing redundant global pseudo-synchronization actions. This corresponds to restoration of the original concurrency and nondeterminacy. In this step, automatic parallelization techniques of the compiler are utilized to identify the set of dependent statements. Pseudo-synchronization actions which are needed to keep precedence constraints of inter-dependent critical statements must be preserved, and other pseudo-synchronization actions can be removed.

Step 4. Code generation. The final concurrent program is generated from the optimized local scenario graphs and the original program. Based on the optimized local scenario graph, each process is reconstructed

by inserting synchronization statements (such as semaphores and monitors) between the original statements. The inserted synchronization statements force the program to behave in the same way of the optimized local scenario graphs. Generic source code optimization can be applied afterwards.

3. FORMALIZATION AND PARALLELIZATION ALGORITHM

This section gives a formal definition of the scenario-based hypersequential programming and provides a parallelization algorithm.

3.1. Formalization

The formal definition starts from the definition of Petri Net and the *communicating transition systems* (CTS). Then, the scenario graph is defined and a new equivalence relation called the *scenario graph equivalence* is introduced. The scenario-based hypersequential programming is a problem to synthesize an optimal CTS which is equivalent to the scenario graph specified by a programmer.

Definition 1 (Petri Net). $N=(P, T, F, m_0)$ is a Petri net where:

- $P=\{p_1, p_2,..., p_n\}$ is a finite set of places,
- $T=\{t_1, t_2,...t_m\}$ is a finite set of transitions,
- $F \subset (P \times T) \cup (T \times P)$ is a set of arcs,
- $m_0: P \to \{0, 1, 2,...\}$ is the initial marking, and
- $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.

In this discussion, the conventional transition firing rules change the marking $m: P \to \{0, 1, 2,...\}$ of a Petri net. $m(p)$ is a number of tokens in a place p. Additional notations are defined as follows:

- $\bullet t = \{p \mid (p, t) \in F\}$ and $t \bullet = \{p \mid (t, p) \in F\}$ represent input and output places of a transition t, respectively.
- $m[t\rangle$ denotes that $t \in T$ is enabled at a marking m. $m[t\rangle m'$ represents the transition $t \in T$ from a marking m to a new marking m'. $m_1[\theta\rangle m_k$ if $\theta = t_1 \cdots t_{k-1}$ is a transition sequence which satisfies $m_1[t_1\rangle m_2, m_2[t_2\rangle m_3,..., m_{k-1}[t_{k-1}\rangle m_k$.
- $R(N)=\{m \mid \exists \theta \in T^*.m_0[\theta\rangle m\}$ is a set of reachable markings of Petri net N, where T^* is a set of finite sequences over T.
- $\theta[i] = t \in T$ is the ith element of θ.

- $c(\theta, i) = (t, k)$ if $t = \theta[i]$ and t appears k times in $\theta[1],..., \theta[i]$. $c(\theta, i)$ represents a transition t with a counter k.
- $\theta|_T$ is a projection operator where $\theta|_T = \theta'$ such that $\theta'[i] = \theta[i]$ if $\theta[i] \in T$, $\theta'[i] = \varepsilon$ (empty sequence), otherwise.

Definition 2 (Communicating Transition Systems). CTS $\mathscr{C} = (N, \Psi)$ is a specific Petri net $N = (P, T, F, m_0)$ with a process structure $\Psi = \{(P_1, T_1), (P_2, T_2),..., (P_n, T_n)\}$ where

- (P_i, T_i) is a process. For $1 \leqslant \forall i \leqslant n$, $P_i \subset P$ and $T_i \subset T$.
- $P = P_1 \cup P_2 \cup \cdots \cup P_n$. For $0 \leqslant \forall i < \forall j \leqslant n$, $P_i \cap P_j = \varnothing$.
- $T = T_1 \cup T_2 \cup \cdots \cup T_n$, and some transitions may be included by multiple processes (i.e., $t \in T_i \cap T_j$), which realize process synchronization.
- Each process is a finite state transition system which satisfies the following formula:

$$\sum_{p \in P_i} m(p) = 1 \text{ and } |t\bullet \cap P_i| = |\bullet t \cap P_i| = 1 \text{ for } 1 \leqslant \forall i \leqslant n,\ \forall m \in R(N)$$

Figure 5 shows an example of CTS $\mathscr{C} = (N, \Psi)$ where $\Psi = \{(P_1, T_1), (P_2, T_2)\}$. (P_1, T_1) and (P_2, T_2) are transition systems with one synchronization transition $t_{1\_2} \in T_1 \cap T_2$. Here, a transition represents a program statement and a place represents a local program state (program counter, memories, etc.) of the target concurrent program.

A state space $SS(\mathscr{C})$ generated from CTS $\mathscr{C}$ is formalized as a global state transition system. $SS(\mathscr{C})$ represents all possible behaviors of $\mathscr{C}$.

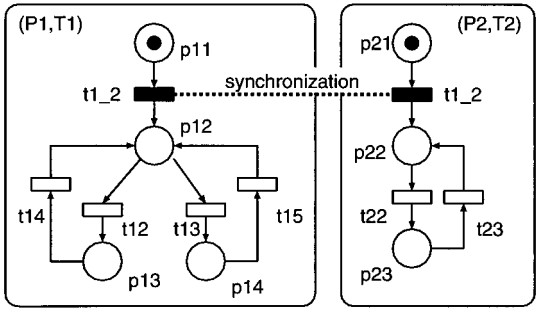


Fig. 5. Communicating transition systems (CTS) $\mathscr{C}$.

Definition 3 (State Space). A state space $SS(\mathscr{C})$ generated from CTS $\mathscr{C} = (N, \Psi)$ is a labeled transition system $SS(\mathscr{C}) = (S, T, \delta, m_0)$ such that

- $S = R(N)$,
- $\delta \subset S \times T \times S$ is a transition relation,
- $\forall m, \forall m' \in S, \forall t \in T.\ (m, t, m') \in \delta$ iff $m[t\rangle m'$

In scenario-based hypersequential programming, a firing transition sequence θ $(m_0[\theta\rangle)$ which the programmer interactively selects is called a *scenario*. The *scenario graph* is constructed from a set of scenarios which can be represented by a labeled transition system (a selected subgraph of the original state space). Every path in the scenario graph is a scenario.

Definition 4 (Scenario Graph). A scenario graph $SG = (S, T, \delta', s_0)$ is a subgraph of a state space $SS(\mathscr{C}) = (S, T, \delta, s_0)$ such that $\delta' \subset \delta$.

Figure 4a is an example scenario graph of the CTS (Fig. 5). In this graph, a node represents a global state (marking), and an edge represents a transition.

A transition corresponds to a program statement such as an assignment statement (e.g., $\mathsf{a} = \mathsf{b} - \mathsf{1}$) or a conditional statement (e.g., $\mathsf{a} == \mathsf{b}$). Concurrent programs with shared memory variables require to consider dependencies between transitions accessing shared variables. Let $D(\mathscr{C})$ be a dependence relation in the CTS $\mathscr{C}$. $(t_i, t_j) \in D(\mathscr{C})$ represents a dependence between t_i and t_j. $D(\mathscr{C})$ is a symmetric (commutative) relation. For a given transition sequence θ, a precedence constraint relation $(\prec)$ among transitions with counters $c(\theta, i) = (t, k)$ in θ is defined according to the transition dependency.

Definition 5 (Precedence Constraint). For a given firing sequence θ of CTS $\mathscr{C}$, $c(\theta, i) \prec c(\theta, j)$ iff $(\theta[i], \theta[j]) \in D(\mathscr{C})$ and $i < j$.

Based on the precedence constraint relation, the equivalence relation between scenario graphs ($\approx_s$: *scenario graph equivalence*) is defined. The scenario graph equivalence is an extended trace equivalence[5] considering precedence constraints. In the definition of trace equivalence, SG_1 and SG_2 are trace equivalent if SG_1 and SG_2 have the same set of scenarios (traces). In the following definition of scenario equivalence, precedence constraints must be equivalent but the order of transitions can be ignored if these transitions are independent (e.g., $ab \approx_s ba$ when a and b are independent transitions) Adoption of bisimulation equivalence instead of trace equivalence is unnecessary since a scenario graph is a *deterministic* transition system.

Definition 6 (Scenario Equivalence). For given scenarios θ_1 and θ_2, $\theta_1 \approx_s \theta_2$ iff

- $\forall i.\exists j.c(\theta_1, i) = c(\theta_2, j)$ and $\forall j.\exists i.c(\theta_2, j) = c(\theta_1, i)$
- $\forall i_1, i_2.\exists j_1, j_2.(c(\theta_1, i_1) = c(\theta_2, j_1)$ and $c(\theta_1, i_2) = c(\theta_2, j_2)$ and $c(\theta_1, i_1) \prec c(\theta_1, i_2) \Rightarrow c(\theta_2, j_1) \prec c(\theta_2, j_2))$
- $\forall j_1, j_2.\exists i_1, i_2.(c(\theta_1, i_1) = c(\theta_2, j_1)$ and $c(\theta_1, i_2) = c(\theta_2, j_2)$ and $c(\theta_2, j_1) \prec c(\theta_2, j_2) \Rightarrow c(\theta_1, i_1) \prec c(\theta_1, i_2))$

Definition 7 (Scenario Graph Equivalence). For given scenario graphs $SG_1 = (S_1, T_1, \delta_1, s_{01})$ and $SG_2 = (S_2, T_2, \delta_2, s_{02})$, $SG_1 \approx_s SG_2$ iff $\forall\theta_1$ in $SG_1.\exists\theta_2$ in $SG_2.\theta_1|_{T_1 \cap T_2} \approx_s \theta_2|_{T_1 \cap T_2}$ and $\forall\theta_2$ in $SG_2.\exists\theta_1$ in $SG_1.\theta_1|_{T_1 \cap T_2} \approx_s \theta_2|_{T_1 \cap T_2}$.

The pseudo-synchronization actions should be ignored in checking scenario equivalence. This is a reason that this definition uses $\theta_1|_{T_1 \cap T_2} \approx_s \theta_2|_{T_1 \cap T_2}$ instead of $\theta_1 \approx_s \theta_2$.

Using the scenario graph equivalence, the scenario-based hypersequential programming can be formalized as follows:

Problem (Scenario Graph Equivalent CTS Synthesis). For a given scenario graph $SG_{\mathscr{C}}$ of CTS $\mathscr{C} = (N, \Psi)$ where $N = (P, T, F, m_0)$ and $\Psi = \{(P_1, T_1),..., (P_n, T_n)\}$, synthesize a new CTS $\mathscr{C}' = (N', \Psi')$ where $N' = (P', T', F', m'_0)$ and $\Psi' = \{(P'_1, T'_1),..., (P'_n, T'_n)\}$ such that

- $T_i \subset T'_i$ for each process (P_i, T_i),
- $SG_{\mathscr{C}} \approx_s SS(\mathscr{C}')$

The synthesized CTS $\mathscr{C}'$ has the same process structure as the original CTS $\mathscr{C}$. However its behavior is restricted by additional pseudo-synchronization actions sync(ID) to guarantee $SG_{\mathscr{C}} \approx_s SS(\mathscr{C}')$. (Figure 6 shows CTS synthesis of the example of Fig. 5.)

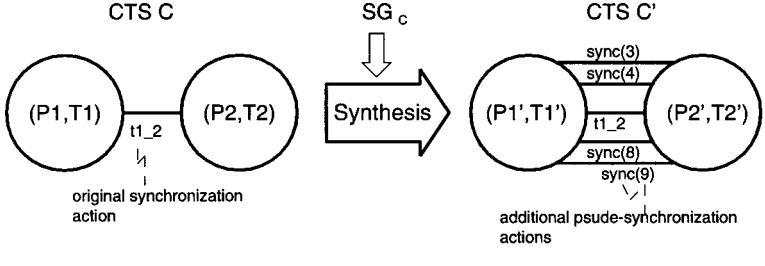


Fig. 6. Scenario graph equivalent CTS synthesis.

There can be multiple scenario-equivalent CTSs. The optimal CTS is the CTS with the fewest pseudo-synchronization transitions are appended. Such a CTS restores the most concurrency and nondeterminacy present in the original program. The measure of optimality is given by the concurrency rate, a simple index of concurrency and nondeterminacy.

Definition 8 (Concurrency Rate). The concurrency rate $\kappa(\mathscr{C})$ of CTS $\mathscr{C}$ is defined as $\kappa(\mathscr{C}) = |\delta|/|S|$ where $SS(\mathscr{C}) = (S, T, \delta, m_0)$.

$\kappa(\mathscr{C})$ is an average number of outgoing edges per state. It indicates how many processes are active and executable concurrently at a typical state.

3.2. Parallelization Algorithm

Based on the formalization, we introduce a concrete parallelization algorithm corresponding to Step 3 of Section 2. The parallelization algorithm consists of 3 steps (introduction of pseudo-synchronization, projection, and optimization). First of all, an optimization algorithm (Step 3.3) for an acyclic CTS is introduced. An acyclic CTS corresponds to a set of local scenario graphs without any loop structures. [Note: A set of local scenario graphs which is generated in Step 3.2 (projection) is straightforward represented by a CTS. We use an CTS instead of a set of local scenario graphs since the CTS is formally defined.] The acyclic-case optimization algorithm constructs $\mathscr{C}_o$ from $\mathscr{C}_n$ (unoptimized CTS) such that $SS(\mathscr{C}_o) \approx_s SS(\mathscr{C}_n)$ and $\kappa(\mathscr{C}_n) \leqslant \kappa(\mathscr{C}_o)$. Before proceeding to the algorithm, a couple of definitions and a theorem need to be introduced.

Definition 9 (Transition Count). For a given finite transition sequence θ, (t, k) is a transition count where the transition t is taken k times in θ.

Definition 10 (Counting Trace). For a given finite transition sequence θ, a counting trace $ct(\theta)$ is:

$$ct(\theta) = (TC\text{set}, PC\text{set})$$

where TCset is a set of all proper transition counts, and PCset is a set of all precedence constraints. A proper transition count is defined as a transition count except pseudo-synchronization.

For example, if $\theta = abcba$ and $(a, c) \in D(\mathscr{C})$, $ct(\theta) = (\{(a, 2), (b, 2), (c, 1)\}, \{c(\theta, 1) \prec c(\theta, 3), c(\theta, 3) \prec c(\theta, 5)\})$.

Definition 11 (Counting Trace Set). For a given acyclic CTS $\mathscr{C} = (N, \Psi)$ and $N = (P, T, F, m_0)$, a counting trace set $ctset(\mathscr{C})$ is:

$$ctset(\mathscr{C}) = \{ct(\theta) \mid m_0[\theta\rangle\, m \wedge \forall t \in T. \neg(m[t\rangle)\}$$

(i.e., θ is a possible and maximal firing transition sequence in $\mathscr{C}$, and m is a terminal (deadlock) state.)

For an acyclic CTS $\mathscr{C}$, the size of $ctset(\mathscr{C})$ is finite.

Theorem 1 (Scenario Graph Equivalence of Acyclic CTS). For given acyclic CTSs $\mathscr{C}_1$ and $\mathscr{C}_2$,

$$SS(\mathscr{C}_1) \approx_s SS(\mathscr{C}_2) \qquad \text{iff} \quad ctset(\mathscr{C}_1) = ctset(\mathscr{C}_2)$$

Proof. When sequences θ_1, θ_2 are finite, $\theta_1 \approx_s \theta_2 \Leftrightarrow ct(\theta_1) = ct(\theta_2)$ by definition. The rest of the proof is trivial.

Definition 12 (Prefix Element of Counting Trace Set). For a given finite transition sequence θ and a given acyclic CTS $\mathscr{C} = (N, \Psi)$ and $N = (P, T, F, m_0)$, $\theta \in_{\text{prefix}} ctset(\mathscr{C}) = (TC\text{set}, PC\text{set})$ iff

- $\forall t \in T.(t, k)$ is a transition count of $\theta \Rightarrow \exists (t, k') \in TC\text{set}.k \leqslant k'$, and
- $\forall i, j.c(\theta, i) \prec c(\theta, j) \Rightarrow c(\theta, i) \prec c(\theta, j) \in PC\text{set}$.

Definition 13 (Deviation Path and Deviation Inductor). For a given acyclic CTS $\mathscr{C}$ and a given transition sequence θ, θ is a deviation path in $\mathscr{C}$ iff $ct(\theta) \notin ctset(\mathscr{C})$. i is a deviation point in the deviation path θ iff $\theta[1] \cdots \theta[i-1] \in_{\text{prefix}} ctset(\mathscr{C})$ and $\theta[1] \cdots \theta[i-1]\, \theta[i] \notin_{\text{prefix}} ctset(\mathscr{C})$. $\theta[k]$ is a deviation inductor iff $\theta[k]$ is a nonsynchronization action, $k \leqslant i$ and $\forall j \leqslant i.\ \theta[j]$ is a nonsynchronization action $\Rightarrow j \leqslant k$, where θ is a deviation path and i is a deviation point.

A deviation inductor means a nonsynchronization action closest to a deviation point in a deviation path.

Algorithm 1 (Optimization Algorithm (Acyclic Case)). For a given acyclic CTS $\mathscr{C}_n$ with pseudo-synchronization actions (Fig. 7b), redundant synchronization actions can be eliminated by the following steps.

Step 1. Insert one local nonsynchronization action nsync(P_i, ID) for each pseudo-synchronization action sync(ID) in each process P_i (Fig. 7c). nsync(P_i, ID) has the same input and output places of sync(ID) but does not synchronize with others. Let this extended CTS be $\mathscr{C}_e$. Let the set of all inserted non-synchronization actions be ANS.

Step 2. Generate a state space $SS(\mathscr{C}_e)$ from $\mathscr{C}_e$ (Fig. 7d).

Step 3. Detect a deviation path θ in $SS(\mathscr{C}_e)$ such that $ct(\theta) \notin ctset(\mathscr{C}_n)$. This indicates some precedence constraints are violated in θ, or a deadlock appears/disappears. Then, find the nonsynchronization action nsync(P_i, ID) which is a deviation inductor (Fig. 7d). The deviation is caused by the introduction of nsync(P_i, ID). Therefore, the synchronization action sync(ID) is necessary (not redundant).

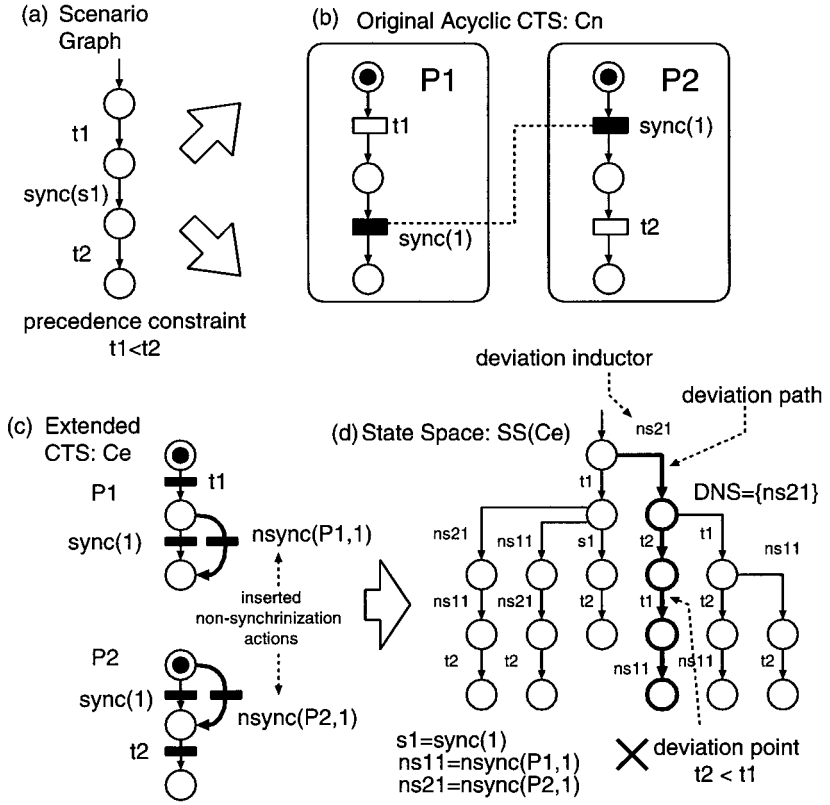


Fig. 7. Simple example of optimization.

Step 4. Let the set of detected nonsynchronization actions be DNS. If $\text{nsync}(P_i, \text{ID}) \in \text{ANS} - \text{DNS}$ then sync(ID) of P_i is redundant. The optimized CTS $\mathscr{C}_o$ is obtained after all redundant synchronization actions are eliminated from $\mathscr{C}_n$. Theorem 1 gives $SS(\mathscr{C}_o) \approx_s SS(\mathscr{C}_n)$.

This optimization algorithm is quite naive and inefficient because it is based on enumeration over the finite state space. It is possible to make it more efficient by introducing several heuristics.

The parallelization algorithm is based on the acyclic optimization algorithm presented earlier. The scenario graph is first partitioned into acyclic subgraphs. Each subgraph is projected onto the original process structure, and local scenario graphs are obtained. Then, they are optimized by Algorithm 1. Finally, optimized local scenario graphs are merged into one. This hierarchical approach is similar to the task-level parallelization approach of Girkar and Polychronopoulos.[6] The parallelization algorithm is shown as follows. It is explained by a simple example (Fig. 8a).

Algorithm 2 (Parallelization Algorithm).

Step 3.1. Preparation.

1. Normalize the scenario graph as a regular expression (Fig. 8b, c). In this expression, each loop is represented as B^*. As in usual regular expressions, "*" represents a repetition of actions and "+" represents alternative actions. An algorithm which transforms a transition system (finite automaton) to an equivalent regular expression is well known.[7]
   Ex. $SG_r = (b_1(a_1a_2a_3)^* a_1(a_2b_2 + b_2a_2) b_3a_3)^*$. SG_r is a regular expression corresponding to a normalized scenario graph (Fig. 8c).
2. Partition the scenario graph into acyclic subgraphs B_k hierarchically. In each hierarchical level, B_k^* is regarded as one action.
   Ex.
   - $SG_r = B_1^*$
   - $B_1 = b_1 B_2^* a_1(a_2b_2 + b_2a_2) b_3a_3$
   - $B_2 = a_1a_2a_3$
3. Insert pseudo-synchronization actions s_i (an abbreviation of sync(i)), after every action and before every branch action and a first action of each block.
   Ex. $B_1 = s_0b_1s_1B_2^*s_2a_1s_3(s_4a_2s_5b_2s_6 + s_7b_2s_8a_2s_9) b_3s_{10}a_3s_{11}$, $B_2 = s_{12}a_1s_{13}a_2s_{14}a_3s_{15}$.

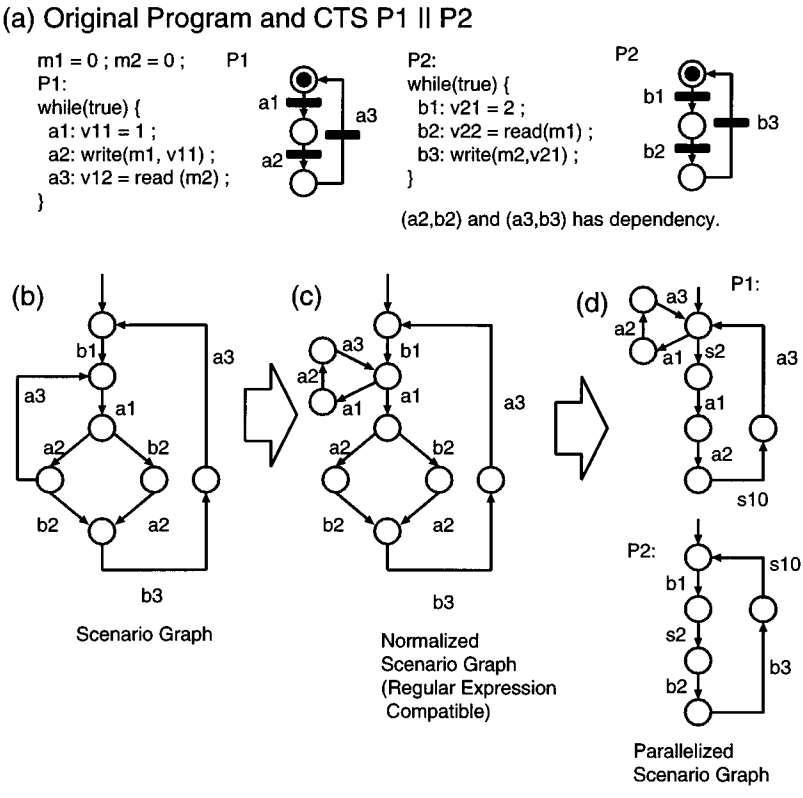


Fig. 8. Example of parallelization.

Step 3.2. Projection. Project the scenario subgraph B_k on to the original process structure. $B_k|_{P_i}$ is a projection of B_k to P_i. $a|_P$ is defined as $a|_P = a$ if a is a P's action, otherwise $a|_P = \varepsilon$. $\theta|_P = \theta[1]|_P \theta[2]|_P \cdots$. Ex. $B_1 = s_0 s_1 (B_2|_{P_1})^* s_2 a_1 s_3 (s_4 a_2 s_5 s_6 + s_7 s_8 a_2 s_9) s_{10} a_3 s_{11} \parallel s_0 b_1 s_1 (B_2|_{P_2})^* s_2 s_3 (s_4 s_5 b_2 s_6 + s_7 b_2 s_8 s_9) b_3 s_{10} s_{11}$, $B_2|_{P_1} = s_{12} a_1 s_{13} a_2 s_{14} a_3 s_{15}$, $B_2|_{P_2} = s_{12} s_{13} s_{14} s_{15}$.

Step 3.3. Optimization.

1. Optimize each acyclic subgraph by removing redundant pseudo-synchronization actions using Algorithm 1.
   Ex. $B_1 = (B_2|_{P_1})^* s_2 a_1 a_2 s_{10} a_3 \parallel b_1 (B_2|_{P_2})^* s_2 b_2 b_3 s_{10}$, $B_2|_{P_1} = a_1 a_2 a_3$, $B_2|_{P_2} = \varepsilon$. "$\parallel$" means parallel composition of processes.
2. Merge the optimized hierarchical acyclic local scenario subgraphs into one local graph per process.
   Ex. $SG_r = ((a_1 a_2 a_3)^* s_2 a_1 a_2 s_{10} a_3 \parallel b_1 s_2 b_2 b_3 s_{10})^*$

3. Then, apply the distribution rule $((B_1 \parallel B_2)^* \Rightarrow B_1^* \parallel B_2^*)$ (Fig. 8d). [Note: We also apply a simple rule when transforming a regular expression back to a transition system.]
   Ex. $SG_r = ((a_1a_2a_3)^* s_2a_1a_2s_{10}a_3 \parallel b_1s_2b_2b_3s_{10})^* \Rightarrow SG_r = ((a_1a_2a_3)^* s_2a_1a_2s_{10}a_3)^* \parallel (b_1s_2b_2b_3s_{10})^*$

Figure 4a is one of the possible scenario graphs $SG_{\mathscr{C}}$ for the CTS $\mathscr{C}$ in Fig. 5. After inserting pseudo-synchronization actions $(s_1, ..., s_{12})$, Figure 4b is obtained (Step 3.1). Naive projection of the graph on to two processes (Step 3.2) produces the two local scenario graphs (Fig. 4c). Suppose $(t_{12}, t_{22}) \in D(\mathscr{C})$ and $(t_{13}, t_{22}) \in D(\mathscr{C})$. Obviously, there are several redundant pseudo-synchronization actions. Since these local scenario graphs have loop structures, Algorithm 2 works as follows.

Step 3.1. Insert synchronization actions.

$$SG = s_0t_{1\_2}s_1(s_3t_{12}s_4t_{22}s_5t_{23}s_6t_{14}s_7 + s_8t_{22}s_9t_{13}s_{10}t_{23}s_{11}t_{15}s_{12})$$

Step 3.2. Project it to the original process structure.

$$SG = s_0t_{1\_2}s_1(s_3t_{12}s_4s_5s_6t_{14}s_7 + s_8s_9t_{13}s_{10}s_{11}t_{15}s_{12}) \parallel$$
$$s_0t_{1\_2}s_1(s_3s_4t_{22}s_5t_{23}s_6s_7 + s_8t_{22}s_9s_{10}t_{23}s_{11}s_{12})$$

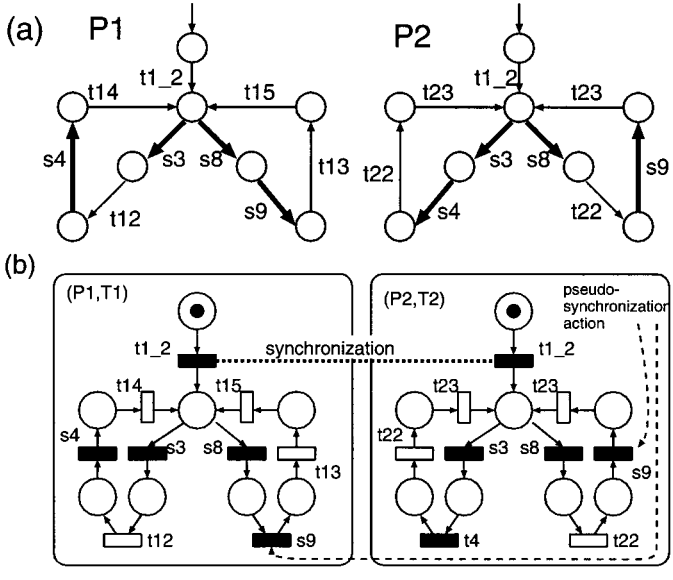


Fig. 9. (a) Optimized local scenario graphs; and (b) their CTS representations.

Step 3.3. Finally, the optimized local scenario graphs are generated by eliminating the redundant synchronization actions $s_0, s_1, s_5, s_6, s_7, s_{10}, s_{11}, s_{12}$ (Fig. 9a).

$$SG = t_{1\_2}(s_3 t_{12} s_4 t_{14} + s_8 s_9 t_{13} t_{15}) \parallel t_{1\_2}(s_3 s_4 t_{22} t_{23} + s_8 t_{22} s_9 t_{23})$$

The CTS $\mathscr{C}_o$ reconstructed from these optimized graphs is scenario equivalent to $SG_{\mathscr{C}}$ (i.e., $SG_{\mathscr{C}} \approx_s SS(\mathscr{C}_o)$) (Fig. 9b).

4. WORKING THROUGH AN EXAMPLE

This section gives a simple and nontrivial example to illustrate how the scenario-based hypersequential programming works. The example uses a shard-memory concurrent program P consisting of two processes P_1 and P_2, and two shared memory variables m_1 and m_2.

Step 1. Modeling and coding. Figure 10 gives the initial program $P = P_1 \parallel P_2$ written by C with function calls to the multi-tasking library in which the semaphores (*signal* and *wait*) are available as synchronization mechanism.

Step 2. Testing scenarios and constructing a scenario graph. Suppose the programmer expects P_2 to print "hello" only after P_1 prints "say hello" and the same for "good-bye" (Fig. 11). If P_2 prints "hello" just after

```
int m1, m2 ; /* Shared Memories */
task P1()
{
  int f ;
  m1 = 0 ;
  m2 = 0 ;
  while(1){
    sleep(10) ;
    f = m1 ;
    if(f==1) {
      printf("Say hello! \n") ;
      f=0 ;
    }
    else {
      printf("Say good-bye! \n") ;
      f=1 ;
    }
    m2 = f ;
  }
}
```

```
task P2()
{
  int f, c ;
  f = 0 ;
  while(1){
    m1 = f ;
    c = m2 ;
    if(c==0) {
      printf("Hello! \n") ;
    }
    else {
      printf("Good-bye! \n") ;
    } ;
    f = c ;
  }
}
```

Fig. 10. Original concurrent program (source code).

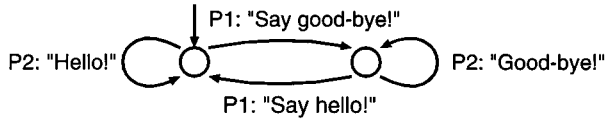


Fig. 11. Programmer's test cases.

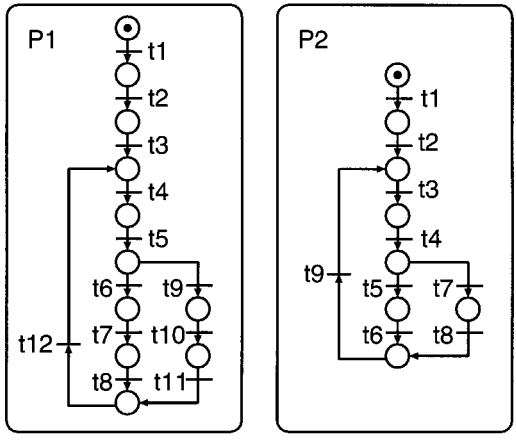
P_1 prints "say good-bye," it is an incorrect behavior. We call this bug the "hello-good-bye-bug."

The programmer executes the program step-by-step, based on the test case, and examines the program behavior. If bugs are detected, the source code must be modified. Suppose the programmer has detected the bug where P_2 accesses m_2 ($\mathsf{c} = \mathsf{m2}$) before m_2 is initialized in P_1 ($\mathsf{m2} = \mathsf{0}$) in the first scenario. Then the programmer can fix it by inserting signal(1) and wait(1). The modified concurrent program and its CTS representation are shown in Figs. 12 and 13, respectively. Then, the programmer executes and tests the modified one. Suppose the programmer has not tested the "hello-good-bye-bug," and it remains in the program as it is. [Note: "hello-good-bye-bug" sequence $= P_2: t_1 \rightarrow P_1 : t_1 \rightarrow P_1 : t_2 \rightarrow P_1 : t_3 \rightarrow P_1 : t_4 \rightarrow P_2 : t_2 \rightarrow P_2 : t_3 \rightarrow P_1 : t_5 \rightarrow P_1 : t_9 \rightarrow P_1 : t_{10} \rightarrow P_2 : t_4 \rightarrow P_2 : t_5 \rightarrow P_2 : t_6$. $P_i : t_j$ is a statement t_j of a process P_i in Fig. 13].

```
int m1, m2 ;
task P1()
{
  int f ;
  m1 = 0 ;
  m2 = 0 ;
  signal(1) ;      /* Inserted */
  while(1){
    sleep(10) ;
    f = m1 ;
    if(f==1) {
      printf("Say hello! \n") ;
      f=0 ;
    }
    else {
      printf("Say good-bye! \n") ;
      f=1 ;
    }
    m2 = f ;
  }
}
```

```
task P2()
{
  int f, c ;
  f = 0 ;
  wait(1) ;      /* Inserted */
  while(1){
    m1 = f ;
    c = m2 ;
    if(c==0) {
      printf("Hello! \n") ;
    }
    else {
      printf("Good-bye! \n") ;
    } ;
    f = c ;
  }
}
```

Fig. 12. Modified concurrent program after fixing one bug.



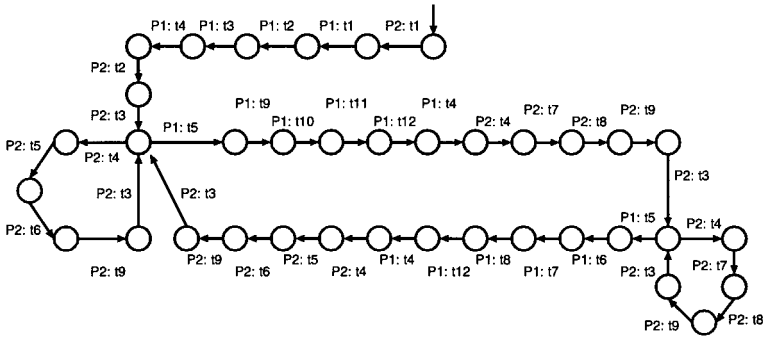

```
P1:
t1: m1=0
t2: m2=0
t3: signal(1)
t4: sleep(10)
t5: f=m1
t6: f==1
t7: print(''Say hello!\n'')
t8: f=0
t9: f==0
t10:print(''Say goodbye! \n'')
t11:f1=1
t12:m2=f1

P2:
t1: f=0
t2: wait(1)
t3: m1=f
t4: c=m2
t5: c==0
t6: print(''Hello! \n'')
t7: c==1
t8: print(''Goodbye! \n'')
t9: f=c
```

Fig. 13. CTS representation $\mathscr{C}$.

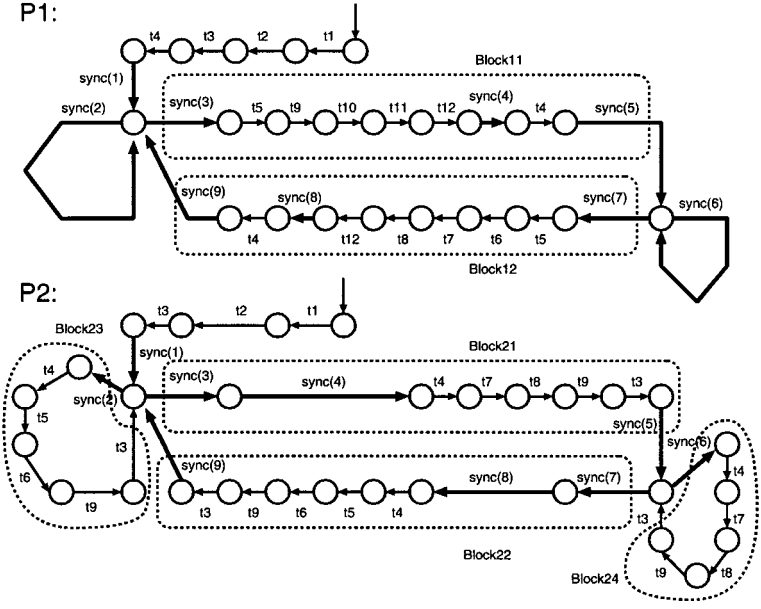
Suppose the programmer has obtained the execution histories which corresponds to the scenario graph $SG_{\mathscr{C}}$ (Fig. 14a). For example, a path $\theta = P_2 : t_1 \rightarrow P_1 : t_1 \rightarrow P_1 : t_2 \rightarrow P_1 : t_3 \rightarrow P_1 : t_4 \rightarrow P_2 : t_2 \rightarrow P_2 : t_3 \rightarrow P_1 : t_5 \rightarrow \cdots$ is a scenario. Since both $P_1 : t_1$ and $P_2 : t_3$ access the shared memory variable m_1 (i.e., $(P_1 : t_1, P_2 : t_3) \in D(\mathscr{C})$), there exists the precedence constraint $c(\theta, 2) = (P_1 : t_1, 1) \prec c(\theta, 7) = (P_2 : t_3, 1)$ in θ. This step of testing and the construction of the scenario graph is supported by the GUI (Fig. 3). Note that this scenario graph does not include an execution

history which produces the "hello-good-bye-bug" since such a sequence is not tested.

Step 3. Parallelization. After inserting pseudo-synchronization transitions `sync(k)`, the scenario graph is divided into the two local scenario graphs. Then, redundant pseudo-synchronization transitions are computed and removed by the parallelization algorithm. Figure 14b shows the optimized local scenario graphs $\mathscr{C}_o$.



(a) Scenario Graph $SG_{\mathcal{C}}$



(b) Optimized Local Scenario Graph $\mathcal{C}_o$

Fig. 14. Scenario graph and optimized local scenario graph.

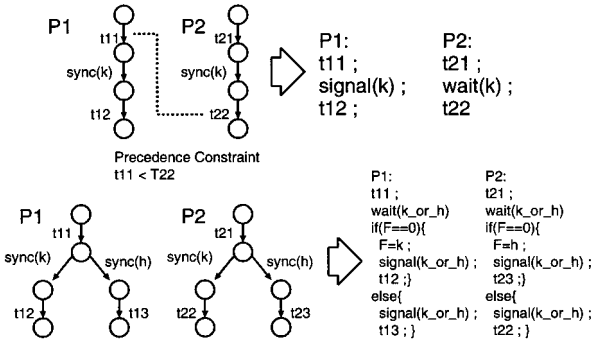


Fig. 15. Code generation pattern.

Step 4. Code Generation. Each process of the concurrent program can be directly reconstructed from the corresponding local scenario graph. Figure 15 gives examples of code generation patterns. Straightforward code generation sometimes causes source code duplications. Some of such duplicated blocks of code can be folded into one. In this example, $Block_{11}$ and $Block_{12}$, $Block_{21}$ and $Block_{22}$, $Block_{23}$ and $Block_{24}$ of Fig. 14 can be folded into single blocks, respectively. [Note: $Block_{11}$ and $Block_{12}$ have duplicated codes (t_5, t_{12}, t_4) and different codes $(t_9, t_{10}, t_{11}, t_6, t_7, t_8)$. Different codes are manipulated by using "if-then-else" structures.]

Figure 16 gives the final concurrent program. The absence of the "hello-good-bye-bug" can be easily observed. The bug has been automatically eliminated from the program even though the programmer has never encountered the bug. Table I gives the concurrency rates for this program. The concurrency and nondeterminacy are restored to 1.68 in the final program from 1.05 of the scenario graph. The original program has the rate of 1.99.

5. RELATED WORKS

Tai *et al.* proposed *deterministic execution debugging* of concurrent Ada programs.[8] This method implements a serialization mechanism based

Table I. Concurrency Rate

	Original program	Scenario graph	Final program
Size (edge/node)	473/237	39/37	411/243
Concurrency	1.99	1.05	1.68

```
int m1, m2, sf ;
task P1()
{
  int f ;
  sf = 0 ;
  signal(4) ;
  m1 = 0 ;
  m2 = 0 ;
  signal(1) ;
  sleep(10) ;
  wait(2) ;
  while(1){
    wait(4) ;
    sf=0 ;
    signal(4) ;
    f = m1 ;
    if(f==1) {
      printf("Say hello! \n") ;
      f=0 ;
    }
    else {
      printf("Say good-bye! \n") ;
      f=1 ;
    }
    m2 = f ;
    signal(3) ;
    sleep(10) ;
    wait(2) ;
  }
}
```

```
task P2()
{
  int f, c ;
  f = 0 ;
  wait(1) ;
  m1 = f ;
  signal(2)
  while(1){
    wait(4) ;
    if(sf == 1) {
      sf = 1 ;
      signal(4) ;
      wait(3) ;
      c = m2 ;
      if(c==0) { printf("Hello! \n") ; }
      else { printf("Good-bye! \n") ; } ;
      f = c ;
      m1 = f ;
      signal(2)
    }
    else {
      signal(4) ;
      c = m2 ;
      if(c==0) { printf("Hello! \n") ; }
      else { printf("Good-bye! \n") ; } ;
      f = c ;
      m1 = f ;
    }
  }
}
```

Fig. 16. Final concurrent program.

on a given execution history (a scenario). Although their serialization scheme is similar to ours, execution histories in their method are used only for testing and debugging while our method also uses them for the reconstruction of the final program. Automatic detection of harmful nondeterminacy from the execution histories has been studied;[9] but the results have not been applied for automatic elimination of such nondeterminacy. Parallelization of sequential programs has been studied extensively over the past 30 years and many automatic parallelization techniques have been developed.[10] You might ask why we do not start with a sequential program instead of a concurrent one that is serialized afterwards. The answer is that the topology of concurrent programs is useful for modeling in many target domains. It would be easier to map natural concurrency on to a concurrent program and start from there than to write a sequential program and automatically parallelize it later. Hypersequential programming preserves the topology of the original program during serialization and restores it during parallelization (Fig. 17) as opposed to extracting parallelism out of the sequential program.

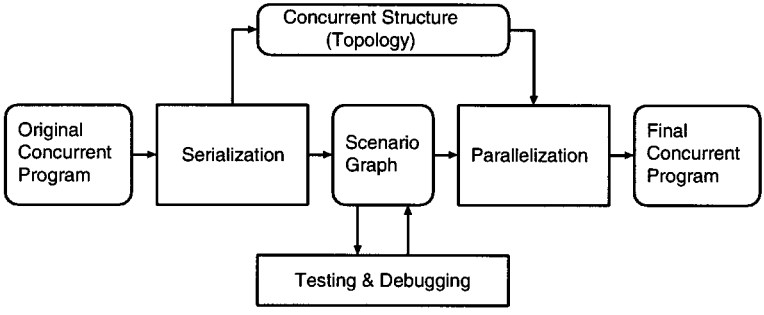


Fig. 17. Hypersequential programming preserves original concurrent structure.

Supervisor control[11] and program adjustment[12] are techniques to automatically modify the program to detect and eliminate harmful nondeterministic behaviors. These approaches require a formal specification to explicitly determine intended behaviors and harmful behaviors. However, it is difficult and impractical to describe a complete formal specification of industrial programs. Hypersequential programming assumes untested behaviors to be harmful. Therefore, it does not require any formal specifications.

6. CONCLUSION

Hypersequential programming can be applied to a wide range of concurrent programming, including parallel programs for shared-memory parallel computers, multi-task programs for embedded systems, and network protocol programs for distributed systems. A CASE tool, based on hypersequential programming, is currently under development. Several heuristics are incorporated to improve the parallelization and optimization algorithm presented in this paper. In order to make the CASE tool practical to use, the following improvements would be needed:

6.1. Global State Abstraction

Examples in this paper assumed all memory states can be represented to identify the global states of the scenario graph. However, a program may access a large amount of memory which is impractical to keep track of. Therefore, some abstraction of global states would be required.

6.2. Hierarchical Scenario

Since it is tedious to make all scenarios at the source code (statement) level, hierarchical and top-down construction of scenarios would be necessary.

Some reader may think hypersequential programming is too radical since there are cases where untested behaviors should not be prevented. In that case, we can also choose the less aggressive approach in which untested actions are only delayed (not prevented) until tested actions happen, and if there is no tested action to happen after a while, untested actions are permitted. This approach can be implemented by slight modification of a hypersequential programming tool. We call this "*timing tranquilizer*." The timing tranquilizer does not eliminate all harmful behaviors, but works fine for many practical cases.

REFERENCES

1. C. M. Pancake, Software Support for Parallel Computing: Where Are We Headed?, *Comm. ACM* **34**(11):53–64 (1991).
2. C. E. McDowell and D. P. Helmbold, Debugging Concurrent Programs, *ACM Computing Surveys* **21**(4):593–622 (1989).
3. N. Uchihira, S. Honiden, and T. Seki, Hypersequential Programming: A New Way to Develop Concurrent Programs, *IEEE Concurrency* **5**(3):44–54 (1997).
4. N. Uchihira and H. Kawata, Scenario-Based Hypersequential Programming: Concept and Example, *Second Int'l. Workshop on Software Enging. Parallel and Distrib. Syst.*, Boston, IEEE Computer Society Press, pp. 277–283 (1997).
5. R. Milner, *Communication and Concurrency*, Prentice-Hall (1989).
6. M. Girkar and C. D. Polychronopoulos, Automatic Extraction of Functional Parallelism from Ordinary Programs, *IEEE Trans. Parallel Distrib. Syst.* **3**(2):166–178 (1992).
7. J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley (1970).
8. K. C. Tai, R. H. Carver, and E. E. Obaid, Debugging Concurrent Ada Programs by Deterministic Execution, *IEEE Trans. Software Enging.* **17**(1):45–63 (1991).
9. P. A. Emrath, S. Ghosh, and D. A. Padua, Detecting Nondeterminacy in Parallel Programs, *IEEE Software* **9**(1):69–77 (1992).
10. H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley (1990).
11. P. J. Ramadge and W. M. Wonham, The Control of Discrete Event Systems, *Proc. IEEE* **77**(1):81–98 (1989).
12. N. Uchihira and S. Honiden, Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE, *J. Syst. Software* **33**(3): 207–221 (1996).