PAPER

Hardware-Software Timing Coverification of Distributed Embedded Systems

Jih-Ming FU[†], **Trong-Yen LEE**[†], ***Regular Members***, **Pao-Ann HSIUNG**[††], ***and*** **Sao-Jie CHEN**[†], ***Nonmembers***

SUMMARY Most of current codesign tools or methodologies only support validation in the form of cosimulation and testing of design alternatives. The results of hardware-software codesign of a distributed system are often not verified, because they are not easily verifiable. In this paper, we propose a new formal coverification approach based on linear hybrid automata, and an algorithm for automatically converting codesign results to the linear hybrid automata framework. Our coverification approach allows automatic verification of real-time constraints such as hard deadlines. Another advantage is that the proposed approach is suitable for verifying distributed systems with arbitrary communication patterns and system architecture. The feasibility of our approach is demonstrated through several application examples. The proposed approach has also been successfully used in verifying deadline violations when there are inter-task communications between tasks with different period lengths.

key words: *hardware-software codesign, distributed embedded systems, linear hybrid automata, coverification, hard deadline*

1. Introduction

Conventionally, hardware design path and software design path are separated in system design cycle. These two design paths remain independent until the stage of system integration. During system integration, if problems are encountered, system designer has to change the hardware and/or software designs, the pay-off is substantial extra cost and even over-due schedule. Hardware-software codesign means meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design [1].

Hardware-software codesign is an emerging field of research that deals with embedded systems having both hardware and software. In past few years, several codesign methodologies were proposed, such as COSMOS [2], TOSCA [3], ECOS project [4], and so on. Codesign tools also abound, such as Ptolemy [5] of UC Berkeley, VULCAN [6] of Stanford University, COSYMA [7] of Braunschweig University, CODES [8] of Siemens, and CoWare of IMEC [9].

From above, most current codesign methodologies or tools *validate* (by simulation or by testing) the codesign results produced, instead of *verifying* them (by formal methods). This is because formal verification often encounters the issue of state-space explosion, which hinders the verification of large, complex systems. We propose a coverification method for hardware-software codesign of distributed embedded systems. Our coverification approach is based on *linear hybrid automata*, because a distributed embedded system can be perceived as a linear system. The theory of hybrid automata was proposed by Alur et al. [10] and has been developed by Henzinger et al. at UC Berkeley as a tool named HyTech [11], which can be used for the automatic analysis of LHA. Our coverification method automatically converts results of codesign to linear hybrid automata, and we use HyTech to perform the automatic verification of real-time constraints, such as deadlines. Another feature is that the proposed method is suitable for verifying distributed systems with arbitrary communication patterns and system architecture. Our contribution mainly lies in the automatic conversion of codesign results into verifiable system model (i.e., linear hybrid automata).

This article is organized as follows. Section 2 describes distributed embedded systems and their behavior. Section 3 gives the formal definition of a linear hybrid automaton and describes our method. Section 4 presents an example illustrating the coverification process. Section 5 concludes the article.

2. Distributed Embedded Systems and Hardware-Software Co-Synthesis

2.1 Distributed Embedded Systems

In recent years, due to the evolution of VLSI technology, the widespread use of computers, and the obvious benefits of installing microprocessors within a system, embedded systems have taken advantage of this trend. Most embedded systems use both off-the-shelf microprocessors and application-specific integrated circuits (ASICs) to implement specialized system functions. The design of an embedded system is unique because it is a hardware-software codesign problem, where hardware and software must be designed together to make sure that the implementation not only functions properly but also meets performance, cost, and relia-

Manuscript received February 2, 2000.
Manuscript revised May 2, 2000.

[†]The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.
[††]The author is with the Institute of Information Science, Academia Sinica, Taipei, Taiwan, R.O.C.

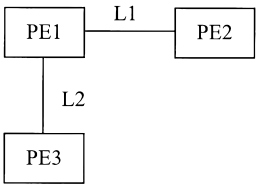


Fig. 1 Processor graph.

Table 1 Computation times of processes $P_1, \ldots, P_5$.

PE type	Computation time				
	P_1	P_2	P_3	P_4	P_5
PE1	299	149	-	203	127
PE2	200	-	128	150	154
PE3	-	184	158	174	180

bility goals [12].

Many embedded systems are distributed systems, with code running in multiple processes on several CPUs/ASICs, and these CPUs/ASICs are connected by communication links. Therefore, we use *processor graphs* to describe these kinds of distributed system architectures. In a processor graph model, each square node represents a processing element (PE) which could be a CPU or an ASIC, and each edge represents a communication link. For example, in Fig. 1, there are three processing elements PE1, PE2, and PE3 and two communication links L1 and L2. PE1 connects to PE2 through L1, and PE1 connects to PE3 through L2. A formal definition of processor graph will be given in Sect. 3.2.

To describe the behavior of a distributed embedded system, we use a *task graph* model [13] to represent functions and performance requirements, without concerning how these functions are implemented, either in hardware or software. Similar task graph model has been used in distributed system scheduling and allocation problems [14]–[20]. The following is an informal description of a task graph model. While its formal definition will be given in Sect. 3.2. The contents of a task graph model are listed as follows:

- Processes: A *process* is the smallest behavior unit of a distributed system, which cannot be interrupted during its execution. Processes can be implemented using either a CPU or an ASIC. Each process is characterized by its computation time, which may not be a constant, but must be bounded for real-time systems. Thus, process computation time is a function of PE (CPU or ASIC). Often a table is used to store these function values. An example is given in Table 1, where an entry of "-" means that the particular process cannot be executed in that type of PE.
- Tasks: A *task* is a partially-ordered set of processes, which is represented as an acyclic directed graph. A directed edge from process Pi to process Pj represents a data dependency, that is, Pj needs to wait for Pi to finish in order to start. Each task must have a START node and an END node. The START node represents the invocation time instant for starting a task. The END node is reached when all processes in a task finish their executions. Worst-case response time is the longest possible elapsed time from START to END for the execution of a task. Each task has a period which is the time between two consecutive invocations of the task, and a hard deadline which is the maximum time allowed from invocation to completion. The period of a task may not be a constant, too. An example of task graphs is illustrated in Fig. 2.
- Data Communication: In a task graph, a weight on a data dependency edge denotes the volume of data emanating from one process to another. We assume that the weights on all edges emanating from the START node are zero and those to the END node are also zero. If the two processes are in two different tasks, the communication is an *inter-task communication*.

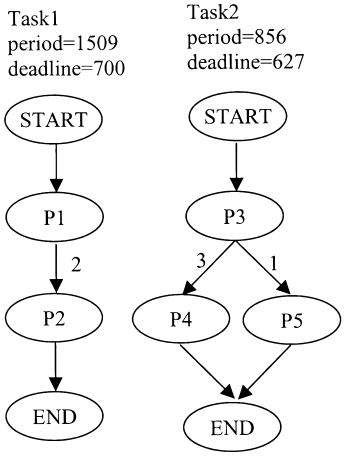


Fig. 2 Example of task graphs.

2.2 Hardware-Software Co-Synthesis

As shown in Fig. 3, hardware-software cosynthesis starts with the specification of an embedded system and results in an architecture of hardware and software modules satisfying the performance, power, and cost goals. Hardware-software cosynthesis involves allocation, scheduling, and performance evaluation. Allocation determines the mapping of processes to processing elements (PEs) and communications to communication links. Scheduling determines the sequencing of processes, their mapping to PEs and the inter-task communication on a link. Performance evaluation determines whether the finally resulting architecture meets all the system specifications. Two distinct approaches have been used for distributed embedded system cosynthesis: optimal and heuristic. Examples in the opti-

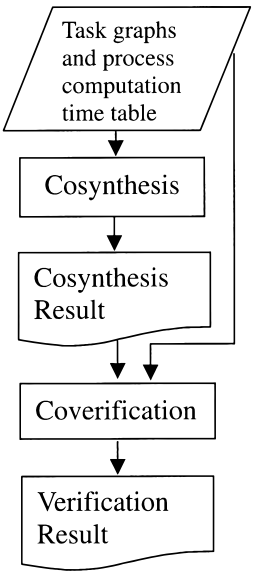


Fig. 3 Overall flow of codesign and coverification.

mal domain are the mixed-integer linear programming (MILP) [17] and the exhaustive [21] approaches. Examples in heuristic domain are the iterative [13], [18], [19], [22] and the constructive (COSYN) [20] approaches.

An overall flow of cosynthesis and coverification is shown in Fig. 3. Our coverification method can be applied to any codesign system result as long as it is modeled using processor and task graphs. For illustration purpose, some codesign results from Yen and Wolf's cosynthesis algorithm [13], [19] are used. Initially, Yen and Wolf's cosynthesis algorithm [13], [19] allocates a PE for each process and a bus for each message. Then, sensitivities are computed, including communication delays and bus cost. *Sensitivity* is the degree by which a system performance and cost will change when a single process is reallocated. An idle-PE elimination technique is applied for the least-utilized bus in the cost calculation. In addition to considering possible reallocation of a process to another PE, possible reallocation of a message to another bus is also considered. When no reallocation remains feasible, a bus is created, in addition to a new PE. The sensitivities for each possible bus type and each message are computed. A reallocation with the highest sensitivity is chosen. After such reallocation is made, communication processes are deleted and regenerated for the new allocation, according to the communication modeling approach [13]. Then, PEs and buses are rescheduled. The above steps are repeated until no reallocation is possible. After such a cosynthesis algorithm, feasible hardware-software systems are produced. In our coverification framework, these design results will be verified to check whether they satisfy deadline constraints.

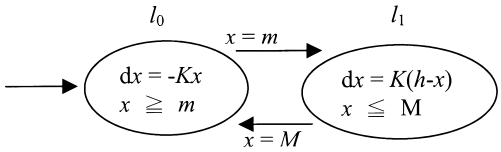


Fig. 4 Thermostat.

3. Coverification Framework

The hardware-software coverification approach proposed in this article is mainly based on the linear hybrid automata model. In this section, hybrid systems are defined and illustrated with examples, along with our coverification method.

3.1 Hybrid Automata Model

Hybrid systems are defined as digital real-time systems that are embedded in analog environments [10]. For example, a thermostat which controls the temperature of a room by sensing the temperature and controlling a heater is a hybrid system. When heater is off, temperature (x) decreases with a rate of $-Kx$; and when heater is on, temperature changes with a rate of $K(h-x)$, where K is a constant related to the room size and h is a constant related to the power of a heater. The specification for the thermostat is that temperature should be maintained between m and M degrees. The hybrid automaton model of a thermostat is shown in Fig. 4. Hybrid systems can also be composed in parallel.

Linear hybrid systems are hybrid systems that have their rate conditions, invariants, and transition relations all linear. In each location of a linear hybrid automaton, the behavior of all variables are governed by linear constraints on the first derivatives [10]. Formal definition of a linear hybrid automaton [23] is shown as follows.

Definition 1: Linear Hybrid Automaton (LHA) A *linear hybrid automaton* (LHA) is a septuple $H = (L, V, B, E, \alpha, \eta, \eta^0)$ such that:

- L is a set of locations,
- V is a set of variables,
- B is a set of synchronization labels,
- E is a set of edges called transitions, $E = \left\{ e \mid e = (l, b, u, l'), l, l' \in L, b \in B, u \in \Phi^2 \right\}$, where Φ is the set of all valuations of variables in V,
- α is a labeling function that assigns to each location a set of rate conditions which are time-invariant,
- η is a labeling function that assigns to each location $l \in L$ an invariant condition $\eta(l) \subseteq V$, and
- η^0 is an initial invariant condition.

Each state in LHA can be denoted by a pair (l, v),

where $l \in L$ and v is the valuation of a variable in V. A run of LHA is a finite or infinite sequence of states

$$\rho : \sigma_0 \rightarrow_{f_0}^{t_0} \sigma_1 \rightarrow_{f_1}^{t_1} \cdots$$

where $t_i \in R^{\geq 0}$, the set of non-negative reals, $f_i \in \alpha(l_i), f_i(0) = v_i, f_i(t) \in \eta(l_i) \forall t, 0 \leq t \leq t_i$ and σ_{i+1} is a transition successor $(\rightarrow_{f_i}^{t_i})$ of $\sigma_i^{'} = (l_i, f_i(t_i))$.

3.2 Coverification Process

The synthesis of a distributed embedded system having hardware and software can be modeled by two kinds of graphs: processor graph and task graph as described in Sect. 2. Processor graph describes system architecture and task graph describes system behavior. According to the task graphs and process characteristics, hardware-software cosynthesis tools [13], [18]–[20] can then be used to generate the topology of a distributed embedded system which meets both the performance and cost goals. But, note that these tools only validate the codesigned system that we have obtained by cosynthesis through simulation, testing, and performance estimation [20]. In order to verify whether a specified distributed embedded system (represented by task graphs and processor graphs) is correct we have to map the specified results into a network of LHAs which can then be used for automatic timing coverification. Therefore, task graphs, process characteristics, and processor graph of a distributed system are the inputs of our coverification algorithm, and a network of LHAs is the output that we need.

Let $\mathcal{N}$ denote the set of non-negative integers, then, we formally define processor and task graphs as follows.

Definition 2: Processor Graph
A processor graph G_P is defined as a tuple $\langle Q_P, T_P, \psi_P, \delta_P, \rho_P, \kappa_P \rangle$, where

- Q_P is a set of nodes representing processors,
- T_P is a set of arcs representing bus links between processors, i.e., $T_P = \{(p, p') \mid p, p' \in Q_P$ and there is a bus link between p and $p'\}$,
- $\psi_P : Q_P \rightarrow 2^{\cup Q_{T_i}}$, $\psi_P(p) = \{q \mid \exists i, q \in Q_{T_i}$ and q is executed on $p\}$, where i is the index label of task T_i,
- $\delta_P : Q_P \rightarrow \mathcal{N}$, $\delta_P(q)$ is the computation rate of processor $q \in Q_P$ and is called the *processor rate*,
- $\rho_P : T_P \rightarrow \mathcal{N}$, $\rho_P(f)$ is the communication rate of bus $f \in T_P$, and
- $\kappa_P : T_P \rightarrow 2^{\cup T_{T_i}}$, $\kappa_P(f) = \{e \mid \exists i, e \in T_{T_i}$ and e is communicated on bus $f\}$, where i is the index label of task T_i.

Definition 3: Task Graph
A task graph G_T is defined as a tuple $\langle Q_T, q_0, T_T, q_f, \chi_T, \tau_T, \pi_T, \delta_T \rangle$, where

- Q_T is a set of nodes representing processes in task T,

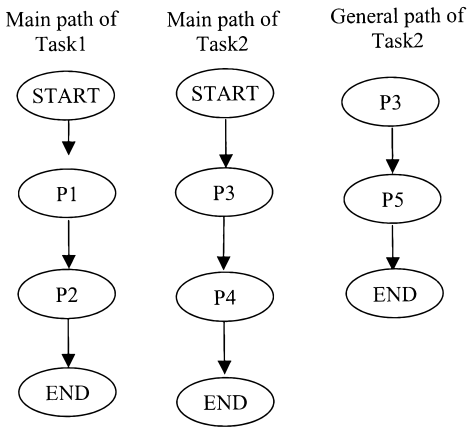


Fig. 5 Example of execution paths.

- $q_0 \in Q_T$ is an initial process of task T,
- T_T is a set of arcs representing communications between two processes, i.e., $T_T = \{(q, q') \mid q, q' \in Q_T$ and process q communicates with (sends data to) process $q'\}$,
- $q_f \in Q_T$ is a final process of task T,
- $\chi_T : Q_T \rightarrow \mathcal{N}$, $\chi_T(q)$ is the execution time of process q on processor $\psi_P^{-1}(q) \in Q_P$,
- $\tau_T : T_T \rightarrow \mathcal{N}$, $\tau_T(e)$ is the volume of data to be transferred for communication e,
- $\pi_T \in \mathcal{N}$ is the period of task T, and
- $\delta_T \in \mathcal{N}$ is the deadline of task T.

We extend the preliminary approach presented in [24] to include communication delay consideration. The new approach presented in this article also reduces the size of LHA network produced. Since a task could have more than one threads or paths of execution that are concurrent, we will extract the concurrent execution paths of each task for transformation into independent LHAs. Note that process (or execution path) *concurrency* in a task can only be modeled by different independent automata. Our algorithm traverses a task graph from its initial START node in a depth-first search manner, such that the first path traversed (and thus executed) is called the *main path* of the task graph. The main path of a task graph always has START as its first node and END as its last node. All other paths obtained in depth-first traversal of the same task graph are called *general paths*. These general paths will not necessarily begin with a START node, nor terminate with an END node. Execution paths for the example in Fig. 2 are illustrated in Fig. 5.

We next provide an overview of our coverification algorithm. Figure 6 presents the pseudo-code of our algorithm. Details will be given in Figs. 7, 8, 9, and 10. First, task graphs, a processor graph, a table of process computation times, and a codesign result are parsed into appropriate data structures. Second, the processor graph is mapped to a network of LHAs. Third, task graphs are mapped to a network of LHAs. Fourth,

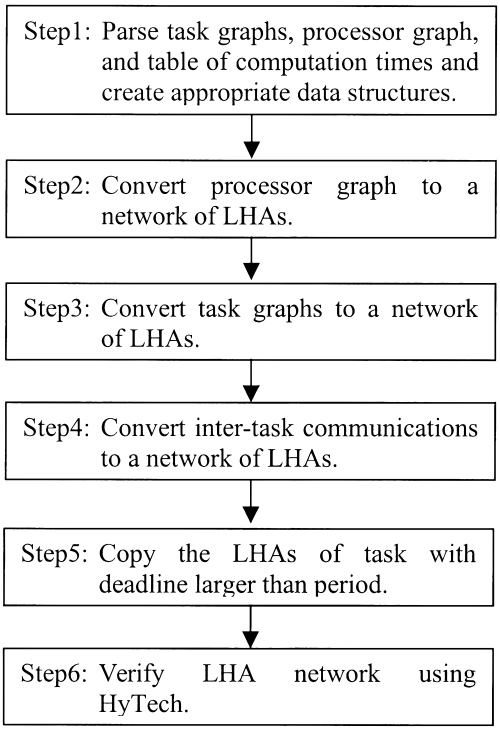


Fig. 6 Coverification algorithm.

LHAs are used to represent inter-task communications. Fifth, copies of LHAs are made for tasks that have deadlines greater than periods since more than one instance of task may be running simultaneously. And finally, we use HyTech to verify LHA network.

Step 1 consists of inputs parsing. First, task graphs, $\{G_{T_i} \mid G_{T_i} = \langle Q_{T_i}, q^0_{T_i}, T_{T_i}, q^f_{T_i}, \chi_{T_i}, \tau_{T_i}, \pi_{T_i}, \delta_{T_i}\rangle, i \geq 1\}$, are parsed and stored in linked lists. Second, a processor graph, $G_P = \langle Q_P, T_P, \psi_P, \delta_P, \rho_P, \kappa_P\rangle$, is parsed into a linked list. Third, a table of process computation times is parsed. Finally, a codesign result, which is to be verified, is input to our algorithm. A codesign result is basically a mapping of each process in each task graph to a processor in the processor graph and a mapping of each communication in each task graph to a bus link in the processor graph. Since each execution of the algorithm verifies a particular codesign result, during the parsing of a table of process computation times, not all computation times from the table are used by the algorithm.

Step 2 consists of a processor graph mapping algorithm as detailed in Fig. 7. Processor graph, G_P, is converted into a network of LHAs $\{H_p \mid H_p = \langle L_p, V_p, B_p, E_p, \alpha_p, \eta_p, \eta^0_p\rangle\} \cup \{H_f \mid H_f = \langle L_f, V_f, B_f, E_f, \alpha_f, \eta_f, \eta^0_f\rangle\}$, where the former represents the set of processors and the latter represents the set of communication buses. An LHA is created for each element (either a node or an edge) in the processor graph. For each node (i.e. processor $p \in Q_P$), a new LHA (H_p) is created, which contains one location

Map_PG(G_P) /* $G_P = \langle Q_P, T_P, \psi_P, \delta_P, \rho_P, \kappa_P\rangle$: a processor graph*/ {
 For each $p \in Q_P$ {
  Create_New_LHA(H_p); /*$H_p = \langle L_p, V_p, B_p, E_p, \alpha_p, \eta_p, \eta^0_p\rangle$*/
  Create_Loc(*Idle*, *true*, $dqt = 0$);
  /* $qt \in V_p$ is a clock and dqt is its rate*/
  $L_p := L_p \cup \{Idle\}$;
  For each $q \in \psi_P(p)$ {
   $\eta^0_p := \eta^0_p \wedge qt = 0$;
   Create_Loc(lq, $0 \leq qt \leq \chi_T(q)$, $dqt = \delta_P(q)$);
   /* $0 \leq qt \leq \chi_T(q) \in \eta_p$, and $dqt = \delta_P(q) \in \alpha_p$.*/
   $L_p := L_p \cup \{lq\}$;
   Create_Arc(a_{q_1}, *Idle*, sSq, $qt := 0$, lq);
   /* lq, $Idle \in L_p$, $sSq \in B_p$, $qt := 0 \Rightarrow \Phi(qt) = 0$, $a_{q_1} \in E_p$*/
   Create_Arc(a_{q_2}, lq, sEq, $qt := 0$, *Idle*);
   /* $sEq \in B_p$, $a_{q_2} \in E_p$*/
  }
 }
 For each $f \in T_P$ {
  if needed(f) = *false*, then **return**;
  Create_New_LHA(H_f); /*$H_f = \langle L_f, V_f, B_f, E_f, \alpha_f, \eta_f, \eta^0_f\rangle$*/
  Create_Loc(*Idle*, *true*, $det = 0$);
  /* $et \in V_f$ is a clock and det is its rate*/
  $L_f := L_f \cup \{Idle\}$;
  For each $e \in \kappa_P(f)$ {
   $\eta^0_f := \eta^0_f \wedge et = 0$;
   Create_Loc(le, $0 \leq et \leq \tau_T(e)$, $det = \rho_P(f)$);
   /* $0 \leq et \leq \tau_T(e) \in \eta_f$, $det = \rho_P(f) \in \alpha_f$
   $L_f := L_f \cup \{le\}$;
   Create_Arc(a_{e_1}, *Idle*, sSe, $et := 0$, le);
   /* le, $Idle \in L_f$, $sSe \in B_f$, $et := 0 \Rightarrow \Phi(et) = 0$, $a_{e_1} \in E_f$*/
   Create_Arc(a_{e_2}, le, sEe, $et := 0$, *Idle*);
   /* $sEe \in B_f$, $a_{e_2} \in E_f$*/
  }
 }
}

Fig. 7 Processor graph mapping.

(l_q) for each of the processes, $q \in \psi_P(p)$, scheduled to be executed by processor p. These locations are called *computation delay locations* and they represent the time delay due to process executions on processors. Further, for each edge (i.e. communication link $f \in T_P$), a new LHA (H_f) is created which contains one location (l_e) for each of the communications $e \in \kappa_P(f)$ that uses edge f. These locations are called *communication delay locations* and they represent time delays due to communications on an edge. Appropriate arcs are then added for modeling the behavior of an actual network of processors. The function procedures Create_Loc() and Create_Arc() used in this algorithm are given in Fig. 9.

Step 3 consisting of a task graph mapping algorithm is detailed in Fig. 8. Each task graph, G_{T_i}, is traversed from the initial START node in a *depth-first search* manner (DFS_Traverse(G_{T_i})). All execution paths, including main path β_0 and general paths β_j, $j \geq 1$, are extracted from G_{T_i}. An LHA, $H_\beta = \langle L_\beta, V_\beta, B_\beta, E_\beta, \alpha_\beta, \eta_\beta, \eta^0_\beta\rangle$, is created for each path of execution $\beta_k \in Paths, k \geq 0$, where $Paths$ is the set of all execution paths for G_{T_i}. The LHA of a main path has locations (DELAY and ERROR) to check pe-

Map_TG(G_{T_i}) /* $G_{T_i} = \langle Q_{T_i}, q^0_{T_i}, T_{T_i}, q^f_{T_i}, \chi_{T_i}, \tau_{T_i}, \pi_{T_i}, \delta_{T_i}\rangle$: a task graph*/ {
 $Paths$ = DFS_Traverse(G_{T_i});
 /* $Paths = \{\beta_0, \beta_1, \beta_2, \ldots\}$, β_0: main path, β_j: general path, $j \geq 1$, and $\beta_j = \langle e_{j0}, e_{j1}, e_{j2}, \ldots\rangle$, $e_{jk} \in T_{T_i}$ */
 For each $\beta_k \in Paths$ ($k \geq 0$) {
  Create_New_LHA(H_β); /*$H_\beta = \langle L_\beta, V_\beta, B_\beta, E_\beta, \alpha_\beta, \eta_\beta, \eta^0_\beta\rangle$*/
  For each edge $e_{kr} \in \beta_k$ /* $e_{kr} = \langle q, q'\rangle, q, q' \in Q_{T_i}$ */ {
   if visited(q) = *true* {
    Create_Loc(Fq, *true*, $dti = 1$); /* $ti \in V_\beta$: a clock */
    Create_Arc(aFq, *last_node*, sFq, NULL, Fq);
    /* *last_node*: last created loc */
   }
   else {
    Create_Loc($Sekr$, *true*, $dti = 1$);
    Create_Arc($aSekr$, *last_node*, $sSekr$, NULL, $Sekr$);
    Create_Loc($Eekr$, *true*, $dti = 1$);
    Create_Arc($aEekr$, $Sekr$, $sEekr$, NULL, $Eekr$);
    if(out_degree(q) > 1) **for**($u = 1, \ldots,$ out_degree(q)) {
     Create_Loc(Fqu, *true*, $dti = 1$);
     Create_Arc($aFqu$, *last_node*, $sFqu$, NULL, Fqu);
    }
   }
   if($\psi_P^{-1}(q) \neq \psi_P^{-1}(q')$) {
    Create_Loc($Sekr$, *true*, $dti = 1$);
    Create_Arc($aSekr$, *last_node*, $sSekr$, NULL, $Sekr$);
    Create_Loc($Eekr$, *true*, $dti = 1$);
    Create_Arc($aEekr$, $Sekr$, $sEekr$, NULL, $Eekr$);
   }
   if(visited(q') = *true* {
    Create_Loc(Jq', *true*, $dti = 1$);
    Create_Arc(aJq', *last_node*, sJq', NULL, Jq');
   }
   else {
    if(in_degree(q') > 1) **for**($u = 1, \ldots,$ in_degree(q')) {
     Create_Loc($Jq'u$, *true*, $dti = 1$);
     Create_Arc($aJq'u$, *last_node*, $sJq'u$, NULL, $Jq'u$);
    }
   }
  }
 }
 Create_DELAY_ERROR(β);
}

Fig. 8 Task graph mapping.

Create_DELAY_ERROR(β) {
 if($\beta = \beta_0$) {
  Create_Loc($JEND$, *true*, $dti = 0$);
  Create_Loc($DELAY$, $ti \leq \pi_{T_i}$, $dti = 0$);
  Create_Loc($ERROR$, *true*, $dti = 0$);
  Create_Arc($aEND1$, *last_node*, $sJEND$, NULL, $JEN\ D$);
  Create_Arc($aEND2$, $JEND$, $ti \leq \pi_{T_i}$, NULL, $DE\ LAY$);
  Create_Arc($aDELAY$, $DELAY$, $ti \leq \pi_{T_i}$, NULL, $START$);
  Create_Arc($aEND3$, $JEND$, $ti \leq \delta_{T_i}$, NULL, $ERROR$);
 }
 else Create_Arc($aEND$, *last_node*, $sJEND$, NULL, $I\ dle$);
}

Let the current LHA be $H = \langle L, V, B, E, \alpha, \eta, \eta^0\rangle$, then the basic functions are defined as follows:

Create_Loc(*Loc_Name, Invariant, Rate_Condition*) {
 Create a location named "*Loc_Name*" $\in L$;
 Assign the invariant condition of *Loc_Name* as *Invariant*, *i.e.*, η(*Loc_Name*) = *Invariant*;
 Assign the rate condition of *Loc_Name* as *Rate_Condition*, *i.e.*, α(*Loc_Name*) = *Rate_Condition*;
}

Create_Arc(*Arc_Name, Source, Trigger, Assign, Destination*) {
 Create an arc named "*Arc_Name*", *i.e.*, *Arc_Name* $= \langle l, b, u, l'\rangle \in E, l, l' \in L, b \in B, u \subseteq \Phi^2$;
 Assign the source location (l) of *Arc_Name* as *Source*;
 Assign the triggering condition (b) of *Arc_Name* as *Trigger*, which might include synchronization label(s) and state-predicates;
 Assign the assignment statements (u) of *Arc_Name* as *Assign*;
 Assign the destination location (l') of *Arc_Name* as *Destination*;
}

Fig. 9 Function calls used in graph mappings.

Map_InterTaskComm($G_{T_1}, G_{T_2}, \ldots$) {
 For each inter-task communication $e = (q, q')$, $\exists i \neq j$ such that $q \in Q_{T_i} \wedge q' \in Q_{T_j}$ {
  Create_Loc($Idle$, *true*, $dtti = 0$);
  Create_Loc(Com, $0 \leq tti \leq \tau_{T_i}(e), dtti = \rho_P^{-1}(e)$);
  Create_Arc(a, $Idle$, sFq, NULL, Com);
  Create_Arc(a', Com, sJq, NULL, Idle);
 }
}

Fig. 10 Inter-task communication mapping.

riod and deadline constraints. Here, we adopt an *edge-oriented* mapping scheme, which is different from the *node-oriented* mapping found in [24]. Then, we visit each edge e_{kr} from the first to the last, and create appropriate locations, transitions, and synchronization labels in the corresponding LHA. Fork locations, Fq and Fqu, are created when a node q has more than one out-going edge (out_degree(q) > 1). These locations synchronize the start of successor processes. Join locations, Jq' and $Jq'u$, are created when a node has more than one in-coming edge (in_degree(q') > 1). These locations synchronize the end of all predecessor processes. Function calls used in task graph mapping, such as Create_DELAY_ERROR(), Create_Loc(), and Create_Arc() are given in Fig. 9.

Step 4 processes inter-task communications, as detailed in Fig. 10. An LHA is created for each inter-task communication, irrespective of whether the two processes involved in an inter-task communication are executed on the same processor or on different processors. Two locations, $Idle$ and Com, and two arcs, a and a', are created for each inter-task communication e between two tasks $q \in Q_{T_i}$ and $q' \in Q_{T_j}$, $i \neq j$.

Step 5 handles cases of tasks, which have deadlines greater than periods. At any moment of execution, a task has at most $\lceil d/p \rceil$ instances at the same time. If a task deadline d is greater than its period p, then the number of instances of the task will be greater than 1. So, we duplicate $\lceil d/p \rceil - 1$ more copies of the LHA produced in Step 3 for the task and change the period to $\lceil d/p \rceil * p$ for all the $\lceil d/p \rceil$ LHA copies. This is required so that all $\lceil d/p \rceil$ instances are active at the same time.

Finally, in Step 6 resulting network of LHAs are input to HyTech for verification.

In the execution of our algorithm, periods are en-

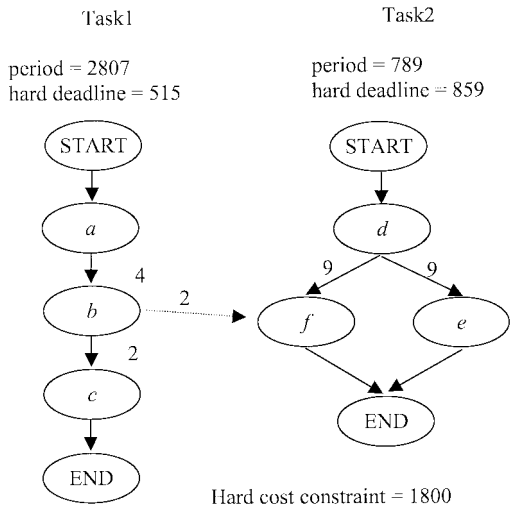


Fig. 11 Task graphs for two tasks, showing periods and deadlines.

forced in an LHA by DELAY locations. Inter-task communications are achieved by HyTech synchronization labels on corresponding transitions between the two LHAs corresponding to the two execution paths with the inter-task communication. Concurrency among different tasks is modeled by assigning a network of LHAs to each task. Concurrency among processes within a single task is also modeled by forking out extra LHAs that represent independent threads of execution in the original task graph. The validity of the above presented algorithm is proved by the following theorem.

THEOREM 1: On verification, if no task LHA enters an ERROR location, all task deadlines will be satisfied.

Proof: When a task LHA did not enter an ERROR location, it should either enter a DELAY location or remain in the *last_node* preceding DELAY. Since all LHAs are assumed to be strongly non-zeno, an LHA cannot remain in the *last_node* forever, hence it will eventually enter DELAY, which implies all deadlines are satisfied.

4. Experimental Results

Details of our coverification algorithm are illustrated through example $ex1$ as shown in Fig. 11, Table 2, and Table 3, which can be found in Yen and Wolf [19]. This example consists of two tasks with periods 2807 and 789 and hard deadlines 515 and 859, respectively, and totally 6 processes a, b, c, d, e, and f as shown in Fig. 11 The dash line from process b to process f is an inter-task communication. The given hard cost constraint was 1800.

This result of hardware-software cosynthesis used three processors X, Y and Z, and two buses of bus type B1, where processes b, c, and f are assigned to processor X, processes d, and e are assigned to processor Y, and process a is assigned to processor Z. There is one bus between X and Z and another between X and Y. The cost of system architecture is 1765, which meets the hard cost constraint. The verification of the above obtained result can be performed as follows.

Table 2 Process computation time and PE cost.

PE type	Cost	Computation time					
		a	b	c	d	e	f
X	800	179	95	100	213	367	75
Y	500	204	124	173	372	394	84
Z	400	210	130	193	399	494	91

Table 3 Communication time per data unit and bus interface cost.

Bus type	Communication time per data unit	Bus interface cost		
		X	Y	Z
$B1$	2	36	19	30
$B2$	1	20	10	15

First, our algorithm transforms the processor graph obtained from the result of hardware-software cosynthesis into a network of linear hybrid automata as illustrated in Fig. 12 and the task graphs in Fig. 11 to a network of linear hybrid automata as illustrated in Fig. 13 and Fig. 14.

Then, in Step 6, this network of LHAs was input to HyTech [11], [25], a popular verification tool for hybrid systems. The real-time constraints such as hard deadlines were verified by checking whether any ERROR location has been reached in the reachability graph produced by HyTech. During HyTech modeling, a scale-down of constants has to be performed for successful verification. Cost factors are ignored since we are considering timing coverification.

Although timed automata could be used for modeling the process execution times in Table 2, yet we use linear hybrid automata because when execution rates of processors are given, LHA could still be used for modeling and verification.

After verification, we have found that the codesigned architecture cannot meet the real-time constraints (hard deadline of 859) for Task 2 because of the periods of tasks being different.

The reason why Task 2 misses its deadline of 859 time units is that process f must wait for data input from process b. The first instants or jobs of the two tasks will complete without violating their respective deadline constraints. But, since the period of Task 1 is 2807 which is much greater than that of Task 2 (789), and since process f must wait for the completion of process b, thus the total execution time of Task 2 will exceed its deadline.

Through our coverification approach, we have shown that the codesign results obtained by Yen and Wolf [19] are not actually feasible when inter-task communication is concerned. This shows the usefulness

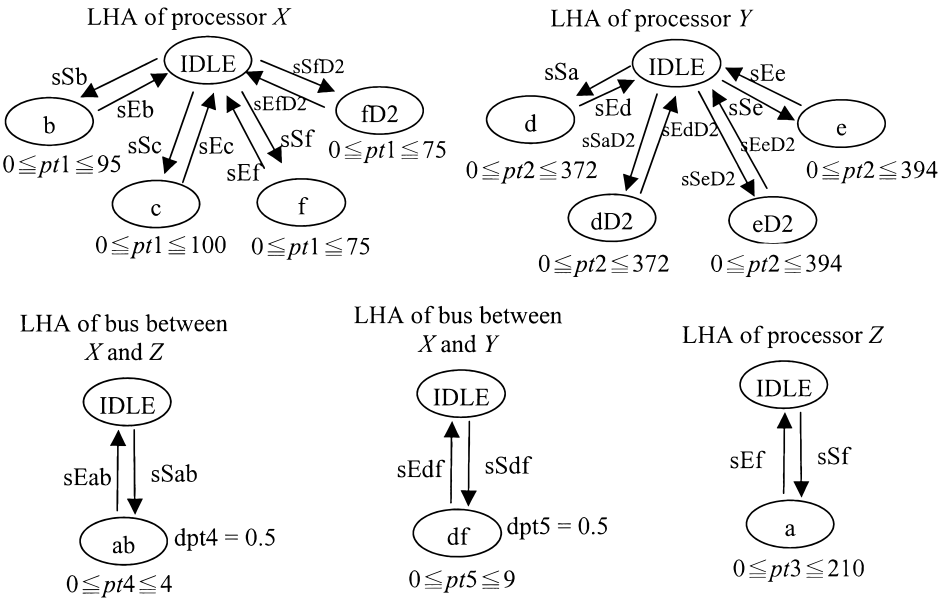


Fig. 12 Networks of LHA for the system architecture obtained in [19].

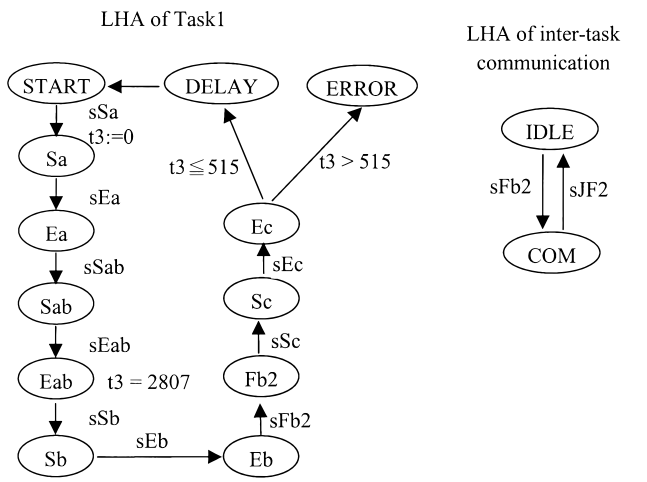


Fig. 13 Networks of LHA for Task 1 and inter-task communication in Fig. 11.

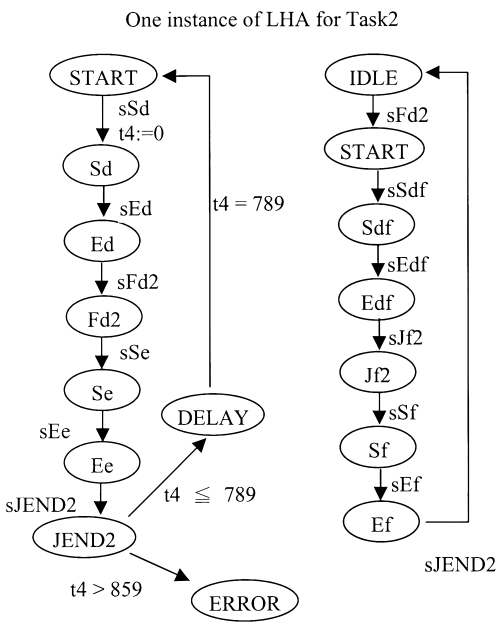


Fig. 14 Networks of LHA for one instance of Task 2 in Fig. 11.

of coverifying design results after hardware-software cosynthesis.

Further, we compare the coverification results obtained using a previously proposed approach [24] and our current approach presented here. The results of running Prakash and Parker's two examples [17] using our previous approach and new approach are given in Table 4. We assume that in Prakash and Parker's first example the periods as well as the deadline are 7, and in the second example they are 15. Prakash and Parker's first example (prakash-parker-e1di) has 4 processes, and 4 design results ($i = 1, \ldots, 4$). But the 4th design result uses only one processor, so we do not convert it as we are only considering distributed systems with more than one processor. The second example (prakash-parker-e2dj) has 9 processes, and 4 design results ($j = 1, \ldots, 4$). As shown in Table 4, example prakash-parker-e1d1 consists of 1 task with 4 processes and the result of synthesis uses 3 processors and 2 links. The result of our previous coverification algorithm showed that we convert the synthesis result into a network of LHAs with 9 automata requiring a total of 1.48 seconds for verification. The result of our new approach proposed in this article shows that we need only a network of LHAs with only 6 automata, and spend 0.28 seconds for verification.

The last column of Table 4 indicates whether the synthesis example satisfies all of its task deadlines.

Table 4 Experimental results: comparison of previous and new methods.

Example	Problem size		Architecture		Previous [24]		New		Deadline Satisfied ?
	#task	#process	#PE	#link	#LHA	Verification time(sec)	#LHA	Verification time(sec)	
*ex*1	2	6	3	2	-	-	11	616.28	No
prakash-parker-e1d1	1	4	3	3	9	1.48	6	0.28	Yes
prakash-parker-e1d2	1	4	3	3	9	1.58	6	0.34	Yes
prakash-parker-e1d3	1	4	2	3	8	1.33	5	0.26	Yes
prakash-parker-e2d1	1	9	3	3	14	143.29	8	3.79	Yes
prakash-parker-e2d2	1	9	3	3	14	157.57	8	3.61	Yes
prakash-parker-e2d3	1	9	2	3	13	103.57	7	2.98	Yes
prakash-parker-e2d4	1	9	2	3	13	98.41	7	2.89	Yes

Table 5 Experimental results: analysis of verification time.

Example	Problem size		Architecture		Coverification		
	# Tasks	# Processes	# PE	# Links	# LHA	Time (sec)	Deadline Satisfied?
T3P3-1	3	11	3	3	8	16.75	Yes
T3P3-2	3	11	3	2	8	30.62	No
T3P4	3	11	4	3	9	53.81	Yes
T4P3	4	15	3	2	10	511.69	No
T4P4	4	15	4	3	11	3187.35	Yes

Here, only the *ex*1 example from [19] does not satisfy its deadline as described above. By comparing the results obtained using the algorithm presented in [24] (Previous in Table 4) and the algorithm proposed in this article (New in Table 4), we conclude that the new algorithm produces a much smaller network of LHAs, which directly reduces the overall verification time. This is especially evident for larger examples such as prakash-parker-e2d*i*. The reduction obtained by the new algorithm in the size of LHA networks generated is as much as 46% and the reduction in verification time is 99%.

To analyze the factors that affect verification time, we applied our approach to some larger examples as shown in Table 5, where T*x*P*y*-*v* indicates that the specified system consists of x tasks, y PEs, and v is an optional label denoting different codesigned architectures. From the table of coverification results, we can deduce that coverification time depends on mainly two factors: the number of PEs and the number of processes. The number of PEs represents degree of *system concurrency* and hence a small change (from 3 PEs to 4 PEs) results in an exponential blow-up in the verification time (compare row 4 and row 5 in Table 5). The number of processes represents degree of *system complexity* and a change (from 11 processes to 15 processes) results in a large difference in the verification time (compare rows 1, 2, 3 with rows 4, 5 in Table 5).

5. Conclusions

An efficient coverification algorithm based on linear hybrid automata has been proposed for hardware-software codesign of distributed embedded systems. This approach automatically converts the results of codesign to linear hybrid automata, thus, it allows automatic coverification of real-time constraints, and is suitable for verifying distributed embedded systems. Experimental results show that the new approach has reduced the number of LHAs and CPU time compared to a previously proposed approach [24] by as much as 46% and 99%, respectively. We have also verified using our method, that the constraint of inter-task communication in a simple codesign example from Yen and Wolf [19] is not feasible.

Acknowledgment

This work was supported by the National Science Council, R.O.C. under grant NSC89-2213-E002-097.

References

[1] G. De Micheli and R. Gupta, "Hardware/software co-design," Proc. IEEE, vol.85, no.3, pp.349–365, March 1997.

[2] T. Ismail and A. Jerraya, "Synthesis steps and design models for codesign," IEEE Computer, no.2, pp.44–52, Feb. 1995.

[3] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto, "The role of VHDL within TOSCA co-design framework," Proc. Euro-VHDL, pp.612–617, Sept. 1994.

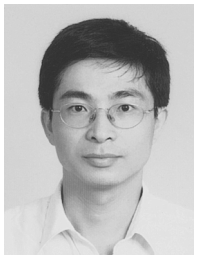
[4] M. Aiguier, J. Benzakki, G. Bernot, S. Berofe, D. Dupont, L. Freund, M. Israel, and F. Rousseau, "ECOS: A generic codesign environment for the prototyping of real-time applications," in Hardware/Software Co-Design and Co-Verification, eds. J.-M. Berge, O. Levia, and J. Rouillard, Kluwer Academic Publishers, 1997.

[5] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," International Journal of Computer Simulation, Special Issue on Simulation Software Development, vol.4, pp.155–182, 1994.

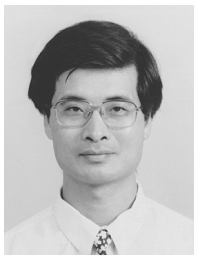
[6] R. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," IEEE Design and Test of Computers, vol.10, no.3, pp.29–41, Sept. 1993.

[7] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for micro-controllers," IEEE Design and Test

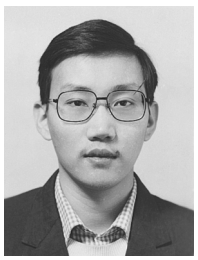
of Computers, vol.10, no.4, pp.64–75, Dec. 1993.
[8] K. Buchenrieder and C. Veith, "CODES: A practical concurrent design environment," Proc. International Workshop on Hardware-Software Co-Design, pp.12–13, 1992.
[9] K. Van Rompaey, D. Verkest, I. Bolsens, and H. De Man, "CoWare — A design environment for heterogeneous hardware/software systems," Proc. Zuropean Design Automation Conference, pp.252–257, 1996.
[10] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," Theoretical Computer Science, vol.138, pp.3–34, 1995.
[11] T. Henzinger, P.-H. Ho, and H. Wong-Toi "A user guide to HyTech," Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNGS, vol.1019, pp.41–71, Springer Verlag, 1995.
[12] W. Wolf, "Hardware-software codesign of embedded systems," Proc. IEEE, vol.82, no.7, pp.967–989, July 1994.
[13] T.-Y. Yen and W. Wolf, Hardware-software co-synthesis of distributed embedded systems, Kluwer Academic Publishers, The Netherlands, 1996.
[14] C. J. Hou and K. G. Shin, "Allocation of periodic task modules with precedence and deadline constraints in distributed real-time system," Proc. Real-Time Systems Symposium, pp.146–155, 1992.
[15] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," Proc. International Conference on distributed Computing Systems, pp.108–115, 1990.
[16] D.-T. Peng and K. G. Shin, "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," Proc. International Conference on Distributed Computing Systems, pp.190–198, 1989.
[17] S. Prakash and A. C. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," Journal of Parallel and Distributed Computing, vol.16, pp.338–351, 1992
[18] T.-Y. Yen and W. Wolf, "Sensitivity-driven co-synthesis of distributed embedded systems," Proc. 8th International Symposium on System Synthesis, pp.4–9, 1995.
[19] T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," Proc. IEEE International Conference on Computer Design, pp.288–294, 1995
[20] B.P. Dave, G. Lakshminarayana, and N.K. Jha, "COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems," IEEE Trans. VLSI Systems, vol.7, no.1, pp.92–104, March 1999.
[21] J. G. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time embedded systems," Proc. International Whokshop Hardware-Software Co-Design, pp.34–41, Sept. 1994.
[22] D. Kirovski and M. Potkonjak, "System-level synthesis of low-power hard real-time systems," Proc. Design Automation Conference, pp.697–702, June 1997.
[23] P.-A. Hsiung, "Timing coverification of concurrent embedded real-time systems," Proc. 7th IEEE/ACM International Workshop on Hardware/Software Co-Design, pp.110–114, ACM Press, New York, USA, May 1999.
[24] J.-M. Fu and S.-J. Chen, "Hardware-software coverification of distributed embedded systems," Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, vol.6, pp.2995–3001, June 1999.
[25] T. Henzinger, P.-H. Ho, and H. Wong-Toi "HyTech: The next generation," Proc. 16th Real-Time Systems Symposium, IEEE Computer Society Press, pp.56–65, 1995.



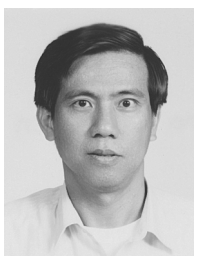
Jih-Ming Fu received the B.S. degree in computer science from the Tamkang University, Taipei, Taiwan, ROC, in 1988. Currently, he is a Ph.D. candidate in the Department of Electrical Engineering, National Taiwan University. His current research interests include distributed real-time system framework, hardware-software cosynthesis, and object-oriented desgin techniques in system synthesis.



Trong-Yen Lee received the B.S. and M.S. degrees from National Taiwan Normal University, Taiwan, Republic of China, in 1981 and 1988, respectively. Currently he is a Ph.D. candidate in the Department of Electrical Engineering, National Taiwan University. His current research interests include parallel architectures, simulation, and design automation systems.



Pao-Ann Hsiung received the B.S. degree in mathematics and the Ph.D. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, ROC, in 1991 and 1996, respectively. From 1993 to 1996, he was a Teaching Assistant and System Administrator in the Department of Mathematics, National Taiwan University. Currently, he is a post-doctoral researcher at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC. His main research interests include: system-level design automation of multiprocessor systems, formal specification, modeling, and verification, parallel architecture design and simulation, and object-oriented design techniques in system synthesis.



Sao-Jie Chen received the B.S. and M.S. degrees in electrical engineering from the National Taiwan University, Taipei, Taiwan, ROC, in 1977 and 1982 respectively, and the Ph.D. degree in electrical engineering from the Southern Methodist University, Dallas, USA, in 1988. Since 1982, he has been a member of the faculty in the Department of Electrical Engineering, National Taiwan University, where he is currently a full professor. From 1985 to 1988, he was on leave from National Taiwan University and working toward his Ph.D. at Southern Methodist University. During the fall of 1999, he was a visiting scholar in the Department of Computer Science and Engineering, University of California, San Diego. His current research interests include: VLSI ciruits design, VLSI physical design automation, object-oriented software engineering, and multiprocessor architecture design and simulation. Dr. Chen is a member of the Chinese Institute of Engineers, the Association for Computing Machinery, the IEEE, and the IEEE Computer Society.