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-
Fig. 1 Processor graph.

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

<table>
<thead>
<tr>
<th>PE type</th>
<th>Computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P_1$</td>
</tr>
<tr>
<td>PE1</td>
<td>299</td>
</tr>
<tr>
<td>PE2</td>
<td>200</td>
</tr>
<tr>
<td>PE3</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 2 Example of task graphs.

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 $P_i$ to process $P_j$ represents a data dependency, that is, $P_j$ needs to wait for $P_i$ 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.

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-
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 verification is shown in Fig. 3. Our verification 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 verification framework, these design results will be verified to check whether they satisfy deadline constraints.

Fig. 3 Overall flow of cosynthesis and verification.

3. Coverification Framework

The hardware-software verification 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 verification 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 \(-K x\); 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, \( E = \{ e \mid e = (l, b, u, l'), l, l' \in L, b \in B, u \in \Phi^2 \} \), where \( \Phi \) is the set of all valuations of variables in \( V \),
- \( \alpha \) is a labeling function that assigns to each location a set of rate conditions which are time-invariant,
- \( \eta \) is a labeling function that assigns to each location \( l \in L \) an invariant condition \( \eta(l) \subseteq V \), and
- \( \eta^0 \) 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
\[
\rho : \sigma_0 \rightarrow^{l_0} \sigma_1 \rightarrow^{l_1} \cdots
\]
where \( t_i \in \mathbb{R}^{\geq 0} \), 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 \( \sigma_{i+1} \) is a transition successor \((\to l_{i+1})\) 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., \( T_P = \{(p, p') | p, p' \in Q_P \} \) and there is a bus link between \( p \) and \( p' \),
- \( \psi_P : Q_P \rightarrow 2^{\mathcal{Q}_T}, \psi_P(p) = \{q | \exists i, q \in Q_T \text{ and } q \text{ is executed on } p\} \), 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
- \( \kappa_P : T_P \rightarrow 2^{\mathcal{Q}_T}, \kappa_P(f) = \{e | \exists i, e \in T_{T_i} \text{ and } e \text{ is communicated on bus } f\} \), 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 \( T \),
- \( q_0 \in Q_T \) is an initial process of task \( T \),
- \( T_T \) is a set of arcs representing communications between two processes, i.e., \( T_T = \{(q, q') | q, q' \in Q_T \text{ and process } q \text{ communicates with (sends data to) process } q'\} \),
- \( q_f \in Q_T \) 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 \( \psi_T^{-1}(q) \in Q_P \),
- \( \tau_T : T_T \rightarrow \mathcal{N}, \tau_T(e) \) 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 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.
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, $\{G_{T_i}|G_{T_i} = (Q_{T_i}, q^0_{T_i}, T_{T_i}, \chi_{T_i}, \tau_{T_i}, \pi_{T_i}, \delta_{T_i})\}$, $i \geq 1$, are parsed and stored in linked lists. Second, a processor graph, $G_P = (Q_P, T_P, \psi_P, \delta_P, \rho_P, \kappa_P)$, 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 $H_P$. 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 $\beta_0$ and general paths $\beta_j$, $j \geq 1$, are extracted from $G_{T_i}$. An LHA, $H_\beta = (L_\beta, V_\beta, B_\beta, E_\beta, \alpha_\beta, \eta_\beta, \eta_0^\beta)$, is created for each path of execution $\beta_k \in \text{Paths}, k \geq 0$, where $\text{Paths}$ 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-
period 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, \( F_q \) and \( F_q^u \), 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, \( J_q' \) and \( J_q'^u \), 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 \text{Create_DELAYERROR()}\), \text{CreateLoc()}\), and \text{CreateArc()}\) 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, \text{Idle} and \text{Com}, and two arcs, \( a \) and \( a' \), are created for each inter-task communication \( e \) between two tasks \( q \in Q_T \) and \( q' \in Q_T, i \neq j \).

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 \( \lceil d/p \rceil \times p \) for all \( \lceil d/p \rceil \) 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-
forced in an LHA by DELAY locations. Inter-task communications are achieved by HyTech synchronization labels on corresponding transitions between the two LHA corresponding to the two execution paths with the inter-task communication. Concurrency among different tasks is modeled by assigning a network of LHA to each task. Concurrency among processes within a single task is also modeled by forking out extra LHA 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 LHA 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.

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 LHA 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 miss 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 verification 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

---

**Table 2** Process computation time and PE cost.

<table>
<thead>
<tr>
<th>PE type</th>
<th>Cost</th>
<th>Computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>X</td>
<td>800</td>
<td>179</td>
</tr>
<tr>
<td>Y</td>
<td>500</td>
<td>204</td>
</tr>
<tr>
<td>Z</td>
<td>400</td>
<td>210</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bus type</th>
<th>Communication time per data unit</th>
<th>Bus interface cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>B1</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>B2</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>
of verifying design results after hardware-software cosynthesis.

Further, we compare the verification 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-e1d) 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-e2d) has 9 processes, and 4 design results ($j = 1, \ldots, 4$). As shown in Table 4, example prakash-parker-e1d consists of 1 task with 4 processes and the result of synthesis uses 3 processors and 2 links. The result of our previous verification 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.

<table>
<thead>
<tr>
<th>Example</th>
<th>Problem size</th>
<th>Architecture</th>
<th>Previous [24]</th>
<th>New</th>
<th>Deadline Satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># task</td>
<td># process</td>
<td>PE</td>
<td>link</td>
<td># LHA</td>
</tr>
<tr>
<td>e.x.1</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>prakash-parker-e1d1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>prakash-parker-e1d2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>prakash-parker-e1d3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>prakash-parker-e2d1</td>
<td>1</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>prakash-parker-e2d2</td>
<td>1</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>prakash-parker-e2d3</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>prakash-parker-e2d4</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5  Experimental results: analysis of verification time.

<table>
<thead>
<tr>
<th>Example</th>
<th>Problem size</th>
<th>Architecture</th>
<th>Coverification</th>
<th>Verification time (sec)</th>
<th>Deadline Satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Tasks</td>
<td># Processes</td>
<td># PE</td>
<td># Links</td>
<td># LHA</td>
</tr>
<tr>
<td>T3P3-1</td>
<td>3</td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>T3P3-2</td>
<td>3</td>
<td>11</td>
<td>3</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>T3P4</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>T4P3</td>
<td>4</td>
<td>15</td>
<td>3</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>T4P4</td>
<td>4</td>
<td>15</td>
<td>4</td>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

Here, only the e.x.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-e2d1. 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 TxyPy-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-E-002-097.

References


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 co-synthesis, and object-oriented design 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 circuits 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.