Timing Coverification of Concurrent Embedded Real-Time Systems

Pao-Ann Hsiung
Institute of Information Science, Academia Sinica, Taipei, TAIWAN.
E-mail: eric@iis.sinica.edu.tw

Abstract

Hardware-software codesign results of concurrent embedded real-time systems are often not easily verifiable. The main difficulty lies in the different time-scales of the embedded hardware, of the embedded software, and of the environment. This rate difference causes state-space explosions and hence coverification has been mostly restricted to the initial system specifications. Currently, most codesign tools or methodologies only support validation in the form of cosimulation and testing. Here, we propose a new formal coverification method based on *linear hybrid automata.* The basic problems found in most coverification tasks are presented and solved. For complex systems, a simplification strategy is proposed to attack state-space explosions in formal coverification. Experimental results show the feasibility of our approach and the increase in verification scalability through the application of the proposed method.

1 Introduction

An *embedded real-time system* is one which is installed in a larger system called *environment.* It is generally a compact, task-oriented, and budget-limited system satisfying timing constraints and cost bounds. Embedded real-time systems usually have both hardware and software interacting with each other to accomplish a specific task. Hardware tries to satisfy timing constraints, and software reduces the overall cost and provides design flexibility. The presence of both hardware and software incurs difficulties in verifying an embedded real-time system. Some common obstacles faced are: the lack of a formal method that can specify both hardware and software, the different time scales of the hardware, the software, and the environment, the requirement of communication protocols between hardware and software, synchronization mechanisms in hardware-software interfaces, and the lack of a formal verification technology devoted to hardware-software coverification. After a careful analysis of possible/existing verification techniques, we felt the need of proposing a new coverification method that can tackle some of the above problems and has is scalable to complex systems.

The three different time scales of an embedded system and its environment posed a great problem in previous approaches (see Section 2). The differing time scales lead to an explosion of state-space during model composition for coverification. *Hybrid automata*, as defined later in Section 3, were proposed for modeling hybrid systems [2]. Not only can each hybrid automaton have a different time scale, but a hybrid automaton can also have different time scales within each location (collection of states). This feature allows the modeling of a multi-rate system that has several timers with different progress rates. In the hardware-software context, this means not only can we model a single chip hardware (1-ASIC) and a uniprocessor software (1-CPU), but also multi-chip hardware (n-ASIC) and multiprocessor software (m-CPU), where $n, m > 0$.

Another reason for using the hybrid automata model is that an embedded digital system can always be perceived as a linear system, that is, the *clock rates* are all linear. The verification theory for linear hybrid automata was proposed by Alur et al in [2] and already implemented in the HyTech tool [11]. Our contribution mainly lies in modeling embedded digital systems using the linear hybrid automata model, demonstrating how basic coverification problems can be solved, experimenting with real examples, and proposing a simplification strategy for coverifying complex systems.

This article is organized as follows. Section 2 describes some related and previous work. Section 3 gives the formal definition of a hybrid automaton and describes how an embedded system can be modeled by a network of hybrid automata. Section 4 presents some elementary commonly-found coverification problems and how they are solved. A simplification strategy is also presented for coverifying complex systems. Section 5 presents an Ethernet Bridge case study illustrating our coverification concepts and method. Section 6 concludes the article with some future work.

2 Previous Work

Large systems can now significantly decrease their overall cost by designing parts of embedded systems as software executing on a general-purpose computation processor. This cost reduction is desirable, but it has also created a few new problems of its own such as the need for a communication protocol between the hardware and software parts, more complicated fault-tolerance problems, the myth that software can be easily *changed*, without any heavy consequences, and coverification problems.

Codesign is an emerging field of research that deals with designing systems that have both hardware and software. In the past few years, several codesign methodologies were proposed, such as COSMOS [8], ECOS project [1], LOTOS-based codesign [15], CMAPS [12] to name a few. Codesign tools also abound, such as SpecSyn [10], Ptolemy, and Polis [4], COSYMA [9], Tyndex, SAW, COWARE, and CHINOOK [7]. Either a combined programming language such as VHDL with C and HardwareC, or some

formal specification language such as LOTOS, ETOILE, Esterel, graphical FSM, and CSP are used for specifying embedded systems. Formal techniques have often been limited to the specification stage such as formal verification of the system specification in LOTOS [15].

From the above, most codesign methodologies or tools currently *validate* the codesigns produced, instead of *verifying* them. Validation occurs in the form of cosimulation and testing. Coverification, although difficult, should not be neglected, especially in high-consequence systems such as nuclear projects, safety systems, etc. The main problems faced in coverifying a design such as different time-scales, etc. were presented in Section 1. Below, we briefly mention two formal models that have been used for coverification and/or codesign, namely CFSM and IPN.

Codesign FSM (CFSM) [5] is a formal model used in the POLIS codesign tool [4]. Coverification is performed by translating CFSM into traditional FSM and existing FSM-based verification techniques applied. The problem of different time scales is not solved because traditional FSM either have no notion of time or their extension such as *Timed Automata* [3] allow specification of clocks with a single uniform rate only. *Intepreted Petri Nets* (IPN) were used for synthesizing interfaces in [16]. Temporal constraints were specified by asserting a delay to a place in IPN. But, the delays occurring in a multi-rate system must be transformed into a common base rate. This transformation is not always ideal or straightforward. Both CFSM and IPN have the same problem of having to handle different time scales, either for coverification or codesign. The hybrid automata model we use for formal coverification solves the problem of different time-scales and at the same time automatic coverification can be performed. We will show how using this model, several coverification problems are solved.

Further, existing real-time system verification tools such as Uppaal [6], SGM (State-Graph Manipulators) [13, 17], and others do not explicitly distinguish hardware and software verification. Since our model is based on hybrid automata, we use the HyTech tool [11] developed by Henzinger, et al. HyTech is a popular tool for verifying hybrid systems.

3 Hybrid Automata Model

The hardware-software timing coverification approach proposed in this article is mainly based on the *hybrid automata model*. There are various reasons for using such a model as given in Section 1. In this section, hybrid systems are defined and illustrated with examples, the hybrid automata model is formally defined, and two different system models for hardware-software coverification are proposed.

The *hybrid automata* model was initially proposed for *hybrid systems*. A hybrid system consists of a discrete program with an analog environment [2]. For example, a thermostat which controls the temperature of a room by sensing the temperature and controlling a heater is a hybrid system because when the heater is off the temperature (x) decreases with a rate of $-Kx$ and when the heater is on, the temperature changes with a rate of $K(h-x)$, where K is a constant related to the room and h is a constant related to the power of the heater. The specification for the thermostat is that the temperature should be maintained between m and M degrees ($0 < m < M$). Other examples of hybrid systems include a water-level monitor, timed mutual-exclusion protocol, leaking gas burner, and a game of billiards [11]. Hybrid systems can also be composed in parallel. *Linear hybrid systems* are hybrid systems that have their activities, invariants, and transition relations all expressed as linear expressions on the system variables [2].

A hybrid automaton can be formally defined as follows.

Definition 1 *Hybrid Automaton (HA)*
A hybrid automaton (HA) is a tuple $H = (L, V, B, E, \alpha, \eta)$, where 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, $E = \{e | e = (l, b, \mu, l'), l, l' \in L, b \in B, \mu \subseteq \mathcal{V}^2\}$, where $\mathcal{V}$ is the set of all valuations of the variables in V, α is a labeling function that assigns to each location a set of *activities* which are time-invariant, and η is a labeling function that assigns to each location $l \in L$ an invariant condition $\eta(l) \subseteq \mathcal{V}$. ‖

A *state* of a hybrid automaton H is a pair (l, v), where $l \in L$ and v is a valuation of the variables in V. A *run* of H is a finite or infinite sequence $\rho : \sigma_0 \rightarrow^{t_0}_{f_0} \sigma_1 \rightarrow^{t_1}_{f_1} \ldots$, where $\sigma_i = (l_i, v_i)$, $t_i \in \mathcal{R}^{\geq 0}$, $f_i \in \alpha(l_i)$, $f_i(0) = v_i$, $f_i(t) \in \eta(l_i) \forall t, 0 \leq t \leq t_i$, and σ_{i+1} is a transition successor of $\sigma'_i = (l_i, f_i(t_i))$.

An embedded system with hardware and software can be mapped into a network of linear hybrid automata (LHA). In the simplest case, one hybrid automaton represents the hardware and one represents the software. The hardware and software interfaces are modeled into the hardware hybrid automaton (HHA) and the software hybrid automaton (SHA), respectively. Another form of modeling could be mapping the hardware part into several LHA each representing some physical hardware component and the software part into several LHA each representing a software process. Due to page-limit, this part is omitted.

4 Coverification Techniques

Using the hybrid automata model for an embedded system, solution techniques are proposed for some commonly-found coverification problems. The five commonly-found elementary coverification problems presented here include: *Software Synchronization*, *Hardware Synchronization*, *Software Concurrency*, *Hardware Concurrency*, and *Integrated Codesign Alternative Verification.* A systematic simplification technique called SHIV (*Software-Hardware-Interface Verification*) is also presented for verifying complex systems. SHIV decomposes the LHA models into three parts, namely the software, the hardware, and the interface, and ensures that the system is safe by performing verification of each part.

4.1 Software Synchronization

In most embedded systems, the software accomplishes some tasks that are costly for the hardware. Often, the hardware makes a request to the software for performing a task and waits for the software to respond. Blocking synchronization is assumed throughout this article because embedded systems are generally synchronous. Asynchrony increases complexity and embedded systems usually cannot afford it. The hardware after making a request waits for a pre-specified period of time, as determined by the system specification or the codesign methodology. If the time limit is reached and the software has not yet responded, the hardware enters a dangerous ambiguous state and the system is unsafe. Coverification must ensure that all such software synchronizations are successful for the given different time scales of the hardware and the software.

Figure 1 shows a LHA simple model of a software synchronization. The hardware has a relative clock rate of $[5/6, 7/6]$ and the software $[3/4, 4/5]$. Running the model using the HyTech tool, we found that software synchronization is guaranteed only if $9h_{max} \geq 14s_{max}$. Further analysis shows that if $[h_l, h_u]$ and $[s_l, s_u]$ were the hardware and software clock rates, respectively, then the condition for software synchronization can be given as a parametric expression:

$$s_l h_{max} \geq h_u s_{min} \quad (1)$$

where h_{max} is the maximum time the hardware, after making a request, will wait for the software response and s_{min} is the maximum time the software must take for computation of the requested task or equivalently the slowest computation delay.

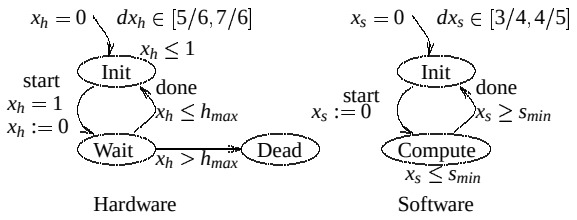

Figure 1: **Software Synchronization**

4.2 Hardware Synchronization

In contrast to software synchronization, hardware synchronization involves a minimum time that the hardware *must* wait after making a request to the software. This situation occurs in the execution of periodic tasks, where the start time of two instances of the same tasks must be separated by a minimum time interval. For example, when the software is responsible for digital signal processing, if two instances of the same tasks overlap randomly, then the computation of the first task will be affected by the second one, thus causing a delay in all future outputs. The situation becomes worse when more than two instances of the same task all overlap causing a heavy workload on the processor executing the software. Coverification in this case must ensure that the hardware does not violate the minimum wait time constraints.

Figure 2 shows a hybrid automata model of a hardware synchronization. The hardware and software relative clock-rate ranges were $[5/6, 7/6]$ and $[3/4, 4/5]$, respectively. The model specification was executed using HyTech and the results obtained: hardware synchronization is guaranteed when the parametric condition $25s_{max} \geq 24h_{min}$ is satisfied. Further analytical study shows that if the hardware and software clock-rate ranges were $[h_l, h_u]$ and $[s_l, s_u]$, respectively, then the hardware synchronization is guaranteed only if the following condition is satisfied.

$$h_l s_{max} \geq s_u h_{min} \tag{2}$$

where s_{max} is the maximum computation time of the software and h_{min} is the minimum wait-time of the hardware.

4.3 Software Concurrency

If a multiprocessor system is within cost constraints for executing the software, a natural question that arises is how many computation processors must be used to speed up software execution in order to cope with hardware requirements and thus guarantee a safe and feasible system. This question can be answered through *Software Concurrency Coverification* (SCC). Software concurrency coverification mainly derives parametric conditions that must be satisfied by a m-processor system ($m \geq 1$) to ensure a safe system.

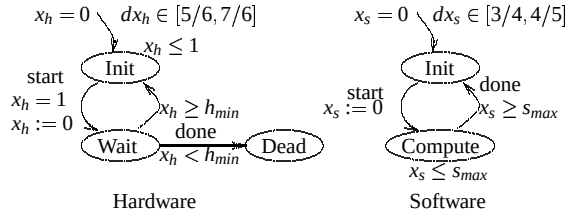

Figure 2: **Hardware Synchronization**

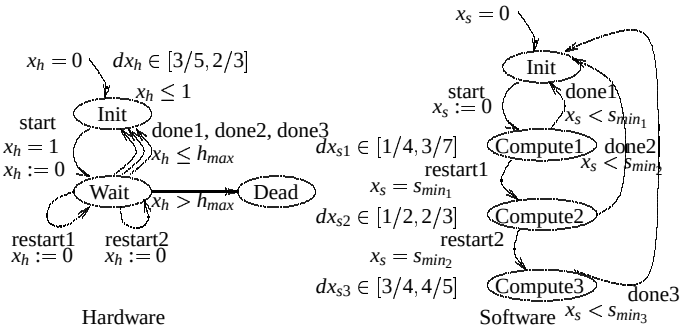

Figure 3: **Software Concurrency**

The clock rates for each configuration of the m-processor system ($m \geq 1$) must be estimated. The hardware waits for some minimum time period after making a request. By increasing the quantity of processors, the software performance could be improved to give results within the hardware minimum time period.

Figure 3 shows a hybrid automata model for a system with one hardware and three possible software configurations: 1-processor, 2-processor, and 3-processor systems. The hardware relative clock rate is assumed to be $[3/5, 2/3]$ and that of the software configurations [1/4, 3/7], [1/2, 2/3], and [3/4, 4/5], respectively. A *sub-linear* increase in computing power of the software configurations is assumed. If h_{max} is the maximum hardware wait-time time and s_{min} is the slowest software computation period, then on running through HyTech either one of the following holds: (1) all the three configurations are safe if $h_{max} > 3s_{min}$, or (2) only the 1-processor system is not safe if $h_{max} > 2s_{min}$, or (3) only the 3-processor system is safe if $h_{max} > s_{min}$.

Depending on the particular task at hand, h_{max} and s_{min} could be estimated and the degree of software concurrency obtained through coverification.

4.4 Hardware Concurrency

In contrast to software concurrency coverification, which increases software performance to meet hardware requirements, *Hardware Concurrency Coverification* (HCC) decreases the hardware cost to meet both the cost and software requirements. Often a cheaper, slower hardware could satisfy all timing requirements in an embedded system. Opting for such a hardware could decrease overall system cost, thus leaving more budget for other embedded systems. Hardware concurrency coverification derives parametric conditions for each hardware-software configuration and the verification engineer could then decide on one particular configuration that meets the timing requirements.

Figure 4 shows the LHA model of hardware concurrency coverification with three hardware configurations H_1, H_2, and H_3 and one software configuration (S). The hardware clock rates are respectively $[1/4, 3/7]$, $[1/2, 2/3]$, and $[3/4, 4/5]$ and that of the software is $[3/5, 2/3]$. Suppose that h_{min_1}, h_{min_2}, and h_{min_3} are the respective minimum time that the hardware configurations must wait (see Hardware Synchronization Coverification in SubSection 4.2) and s_{max} be the maximum computation time of software. Running this model through HyTech, we obtain the result that the system configurations are safe only if the following conditions are satisfied: (1) (H_1, S) is safe if $3h_{min_1} < s_{max}$, (2) (H_1, S) and (H_2, S) are safe if $2h_{min_2} < s_{max}$, and (3) (H_1, S), (H_2, S), and (H_3, S) are all safe if $h_{min_3} < s_{max}$. Hence, if in the slowest and cheapest hardware

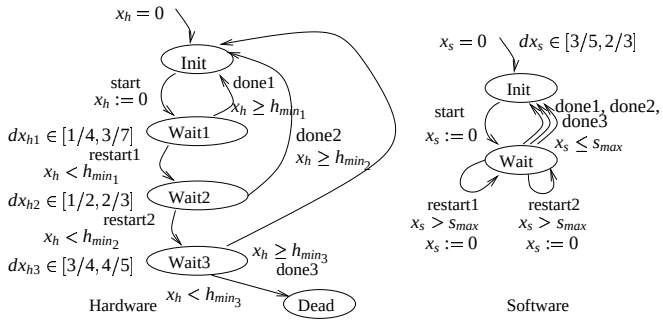

Figure 4: **Hardware Concurrency**

configuration (H_1) the condition $3h_{min_1} < s_{max}$ is met, then we can use H_1 instead of the costlier H_2 and H_3 hardware configurations.

4.5 Integrated Codesign Alternative Verification

Integrated Codesign Alternative Verification (ICAV) handles the case of complex embedded systems with more than one hardware architectures and a multiprocessor system for executing the software. Several codesign alternatives may be produced and validated by a codesign methodology. Normally the selection criterion depends on either the cost (minimum cost) or the performance (maximum throughput) or both (minimum cost-performance ratio). ICAV proposes a new criterion, namely *Incompatibility Ratio of Software-Hardware* (IRSH), which is defined as the *safest* ratio of hardware and software clock rates. By a *safe ratio*, it means that the ratio is either a minimum or a maximum that must be satisfied by an embedded system's clock rates in order for the system to be safe. IRSH is a global *minimum* ratio when there are purely software synchronizations, it is a global *maximum* ratio when there are purely hardware synchronizations, otherwise it is expressed as a range with its lower bound being the minimum of all locally minimal ratios corresponding to software synchronizations and its upper bound being the maximum of all locally maximal ratios corresponding to hardware synchronizations. This metric achieves a better trade-off between the hardware and the software than the conventional cost-performance ratio because the latter can be deceiving at times for very low costs and peak performances.

IRSH is best illustrated by an example as shown in Fig. 5. There are two hardware alternatives with clock rates $[3/2, 15/8]$ and $[5/6, 7/6]$ and two software alternatives with clock rates $[3/4, 4/5]$ and $[1/2, 5/8]$. This example is a case of multiple software synchronization. Table 1 shows the four different configurations (C_1, C_2, C_3, C_4) achievable by the two hardware and the two software alternatives along with their costs, performance values, and cost-performance ratios. We observe that under different metrics the best design configuration is different:

- C_4 has the least cost, but it has a very poor performance,
- C_1 has the best performance, but it has a very high cost,
- C_2 has the best cost-performance ratio, but on applying ICAV we found that it has the largest software-hardware incompatibility (highest IRSH), which means synchronization and other communications could require a large effort, and
- C_3 has the least IRSH, which means that the hardware and the software are the least incompatible and thus achieves a better hardware-software trade-off than the others.

Table 1: **ICAV Example**

Conf	HW Clock	SW Clock	Cost	Perf	Cost/Perf	IRSH
C_1	$[3/2, 15/8]$	$[3/4, 4/5]$	1000	**100**	10.00	2.5
C_2	$[3/2, 15/8]$	$[1/2, 5/8]$	750	80	**9.38**	3.75
C_3	$[5/6, 7/6]$	$[3/4, 4/5]$	650	60	10.83	**1.56**
C_4	$[5/6, 7/6]$	$[1/2, 5/8]$	**500**	50	10.00	2.33

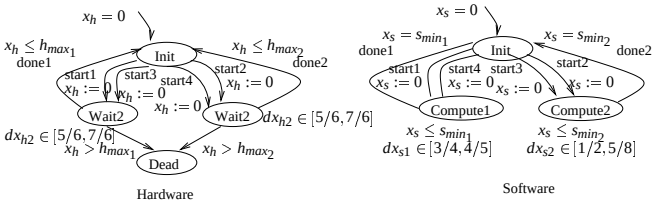

Figure 5: **Integrated Codesign Alternative Verification**

4.6 Software-Hardware-Interface Verification

A new modularized verification strategy called *Software-Hardware-Interface Verification* (SHIV) is proposed for hardware-software embedded systems. Generally, the software and the hardware of an embedded system communicate either through an interface using communication protocols or through shared memory using synchronization variables. The interface is often explicit and important in an embedded system. The SHIV strategy verifies an embedded system by verifying each part individually, namely the hardware, the software, and the interface. The *assume-guarantee principle* of formal modular verification [14] is employed in SHIV. In verifying (*guaranteeing*) the interface, it is *assumed* that both the hardware and the software themselves are correct. Similarly, the principle is applied to the other two parts: the hardware and the software.

In the context of the linear hybrid automata model, SHIV must perform each of the following steps to verify a system.

- *Software Verification*: The triggering conditions on the transitions interconnecting the interface and the software are assumed to be TRUE. All clock variables are either reset or advanced a period of time depending on the triggering conditions on the above transitions.
- *Hardware Verification*: The triggering conditions on the transitions interconnecting the interface and the hardware are assumed to be TRUE. All clock variables are either reset or advanced a period of time depending on the triggering conditions on the above transitions.
- *Interface Verification*: The triggering conditions on the transitions interconnecting the interface and the hardware and on the transitions interconnecting the interface and the software are assumed to be TRUE. All clock variables are either reset or advanced a period of time depending on the triggering conditions on the above transitions.

5 Ethernet Bridge Case Study

Besides the five elementary problems presented in the previous section, we had applied our approach to several real-world systems. An Ethernet Bridge [15] example is presented in this section for illustration. It is assumed as in [15] that the Ethernet LANs oper-

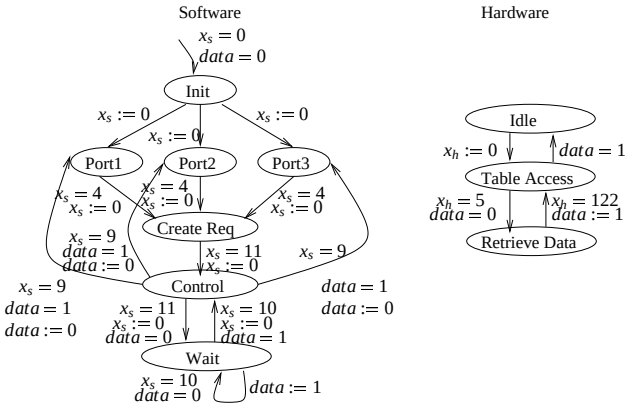

Figure 6: **Ethernet Bridge: Hardware/Software Models**

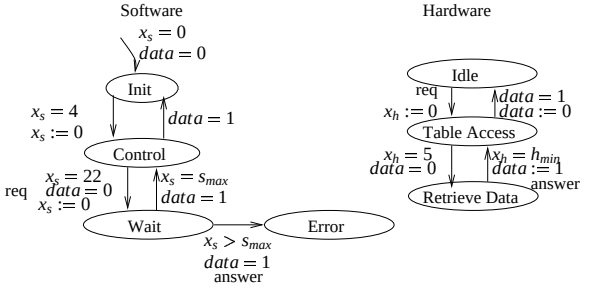

Figure 7: **Ethernet Bridge: Interface Models**

ate under CSMA/CD. The communication estimates given in [15] were transformed into our linear hybrid automata model.

It was found that if the LHA model was directly verified using HyTech, it could not terminate even after modifying the system model as indicated in the HyTech user guide [11]. Finally, the SHIV strategy was applied. The decomposed hardware and software LHA are shown in Fig. 6 and the interface LHA in Fig. 7. We present the interface verification which is the most important for a codesign problem. The bridge processing rate was 3000 pps (packets per second) and the hardware area was 4000 [15].

Given a hardware clock range of $[51/10, 6]$ and a software clock range of $[1/5, 2/5]$, the safety condition was $4h_{min} \leq 51s_{max}$. For the estimates found in [15], h_{min} is 127 and s_{max} is 10, hence the condition is satisfied.

Since the above condition depends on the clock rates, a further analysis shows that if $[h_l, h_u]$ and $[s_l, s_u]$ were the respective hardware and software clock ranges, then the condition would be:

$$\frac{h_{min}}{s_{max}} \leq \frac{h_l}{s_u} \quad \text{or} \quad s_u h_{min} \leq h_l s_{max} \tag{3}$$

6 Conclusion

A linear hybrid automata model based coverification approach was proposed for hardware-software embedded systems. It was shown how different time scales of the hardware and the software and the environment could be handled by the model. Five commonly-found elementary coverification problems were presented and solved using the proposed approach. A simplification strategy called SHIV was also proposed for complex systems. Finally, an Ethernet Bridge case study was presented which showed how SHIV could be used to verify a system when the traditional approach failed. Future work will include developing more strategies using the linear hybrid automata model to solve other coverification problems.

References

[1] M. Aiguier, J. Benzakki, G. Bernot, S. Beroff, D. Dupont, L. Freund, M. Israel, and F. Rousseau. ECOS: A generic codesign environment for the prototyping of real-time applications. In J-M. Berge, Oz Levia, and Jacques Rouillard, editors, *Hardware/Software Co-Design and Co-Verification.* Kluwer Academic Publishers, 1997.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[3] R. Alur and D. Dill. Automata for modeling real-time systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[5] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the Design Automation Conference*, 1996.

[6] J. Bengtsson, K. Larsen, F. Larsson, P. Petterson, Y. Wang, and C. Weise. New generation of UPPAAL. In *Procs. of the International Workshop on Software Tools for Technology Transfer*, July 1998.

[7] P.H. Chou, R.B. Ortega, and G. Borriello. The CHINOOK hardware-software co-synthesis system. In *Procs. International Symposium on System Synthesis*, 1995.

[8] J.M. Daveau, G.F. Marchioro, T. Ben-Ismail, and A.A. Jerraya. COSMOS: An SDL based hardware/software codesign environment. In J-M. Berge, Oz Levia, and Jacques Rouillard, editors, *Hardware/Software Co-Design and Co-Verification.* Kluwer Academic Publishers, 1997.

[9] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for micro-controllers. *IEEE Design and Test of Computers*, 10(4), December 1993.

[10] D. Gajski, F. Vahid, and S. Narayan. A design methodology for system specification refinement. In *Procs. European Design Automation Conference*, February 1994.

[11] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In *Procs. Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, volume 1019, pages 41–71. Springer Verlag, 1995.

[12] P.-A. Hsiung. CMAPS: A cosynthesis methodology for application-oriented parallel systems. *ACM Trans. on Design Automation of Electronic Systems*, 5(2):to appear, April 2000.

[13] P.-A. Hsiung and F. Wang. A state-graph manipulator tool for real-time system specification and verification. In *Proc. 5th. IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, October 1998.

[14] Orna Kupferman and M.Y. Vardi. On the complexity of branching modular model checking. In *Procs. 6th International Conference on Concurrency Theory, LNCS*, volume 962, August 1995.

[15] L. Sanchez, M. L. Lopez, N. Martinez, C. Carreras, J.C. Lopez, C. Delgado-Kloos, A. Royo, and P.T. Breuer. Co-design at work: The ethernet bridge case study. In J-M. Berge, Oz Levia, and Jacques Rouillard, editors, *Hardware/Software Co-Design and Co-Verification.* Kluwer Academic Publishers, 1997.

[16] Christopher Vial and Bruno Rouzeyre. Hardware-software co-synthesis: Modelling and synthesis of interfaces using interpreted petri nets. In J-M. Berge, Oz Levia, and Jacques Rouillard, editors, *Hardware/Software Co-Design and Co-Verification.* Kluwer Academic Publishers, 1997.

[17] F. Wang and P.-A. Hsiung. Automatic verification on the large. In *Proc. 3rd IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, pages 134–141, November 1998.