Hardware-software timing coverification of concurrent embedded real-time systems

P.-A.Hsiung

Abstract: The results of hardware-software codesign of concurrent embedded real-time systems are often not verified or not easily verifiable. This has serious consequences when high-assurance systems are codesigned. The main difficulty lies in the different time-scales of the embedded hardware, of the embedded software, and of the environment. This difference makes hardware-software timing coverification not only a difficult task for most systems, but has also restricted coverification to the initial system specifications. Currently, most codesign tools or methodologies only support validation in the form of cosimulation and testing of design alternatives. Here, a new formal coverification approach is proposed based on *linear hybrid automata*. The basic timing problems found in most coverification tasks are presented and solved. For complex systems, a simplification strategy is proposed to attack the state-space explosion occurring in formal coverification. Experimental results show the feasibility of the 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 within a larger system called its *environment*. It is generally a compact, task-oriented, and budget-limited system. It has to satisfy timing constraints as well as cost limits. Hence, 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, synchronisation mechanisms in hardware-software interfaces, and the lack of a formal verification technology devoted to hardware-software coverification. After careful analysis of possible verification techniques and a survey of existing approaches, the need was felt of a new coverification method that can tackle some of these problems and at the same time has the potential of scaling to industrial processes.

Coverification is defined as formally verifying if a hardware-software embedded system satisfies prespecified real-time properties. A popular technology for verifying real-time systems is called *model checking*. Model checking [1, 2] is an algorithmic procedure for verifying if a real-time system satisfies given temporal properties. A real-time system is often modelled by *timed automata* (TA) [3] and temporal properties are specified using *timed computation tree logic* (TCTL) [2].

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 in Section 3, were proposed for modeling hybrid systems [4]. 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 modelling of a multirate system that has several timers with different progress rates. In the hardware-software context, this means not only can one model single-chip hardware (1-ASIC) and a uniprocessor software (1-CPU), but also multichip hardware (n-ASIC) and multiprocessor software (m-CPU), where $n > 0$ and $m > 0$.

It is well-known that a protocol or any other control-related system is best modelled by *finite state machines* (FSM) [5]. The states and transitions occurring in protocols or controllers can be explicitly and formally specified by FSM. The theory of formal verification has a large part based on FSM. Further, hardware-software systems either require communication protocols for message-passing or shared memory for synchronisation. For these two reasons, if a complete hardware-software system is modeled by FSM, there is no need of specifying the interfaces separately. Hybrid automata are just another extension of FSM.

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 [4] and already implemented in the HyTech tool [6]. Our contribution mainly lies in modelling embedded digital systems using the linear hybrid automata model, demonstrating how basic coverification problems

© IEE, 2000
IEE Proceedings online no. 20000452
DOI: 10.1049/ip-cdt:20000452
Paper first received 30th September 1999 and in revised form 29th February 2000

The author is with the Institute of Information Science, Academia Sinica, Taipei, Taiwan
E-mail: hpa@computer.org

can be solved, experimenting with real examples, and proposing a simplification strategy for coverifying complex systems.

2 Previous work

In recent years, due to computer technology evolution, the widespread use of computers, and the obvious benefits of installing a computing processor within a system, embedded systems have taken advantage of this trend. 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 timing 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 [7], TOSCA [8], ECOS project [9], LOTOS-based codesign [10], and CMAPS [11], to name a few. Codesign tools also abound, such as SpecSyn [12], Ptolemy, and Polis [13] all three of UC Berkeley, VULCAN [14] of Stanford University, COSYMA [15] of Braunschweig University, CODES [16] of Siemens, Tyndex of INRIA, SAW of CMU, COWARE of IMEC, and CHINOOK [17] of Washington University. Either a combined programming language such as VHDL with C and HardwareC, or some formal specification language such as LOTOS, ETOILE, Esterel, graphical FSM, CSP, etc 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 [10].

From this, most codesign methodologies or tools currently *validate* the codesigns produced, instead of *verifying* them. Validation occurs in the form of emulation, 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 time-scale disparity, etc. were presented in Section 1.

Compared with the successful application of formal methods to hardware design [18–21] and verification, there has been little efforts on formalising hardware-software codesign and coverification. Two formal models that have been used for coverification and/or codesign are codesign FSM and interpreted Petri nets.

Codesign FSM (CFSM) [22–24] is a formal model used in the POLIS codesign tool [13]. 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 synthesising interfaces in [25]. Temporal constraints were specified by asserting a delay to a place in IPN. But the delays occurring in a multirate 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 used for formal coverification solves the problem of different time-scales and at the same time automatic coverification can be performed. Using this model, several coverification problems are solved. Further, existing real-time system verification tools such as Uppaal [26], SGM (state-graph manipulators) [27–29], and others do not explicitly distinguish between hardware verification and software verification. Since the model is based on hybrid automata, the HyTech tool [6] developed by Henzinger, *et al.* is employed. HyTech is a popular tool for verifying hybrid systems and is described in greater detail in Section 3.4.

3 Hybrid automata model

The hardware-software timing coverification approach proposed in this article is based mainly on the *hybrid automata model.* The various reasons for using such a model were 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. Parametric analysis and the HyTech tool is also introduced.

The *hybrid automata* model was initially proposed for *hybrid systems.* A hybrid system consists of a discrete program with an analogue environment [4]. For example as shown in Fig. 1, 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 [30, 6]. Hybrid systems can also be composed in parallel.

A hybrid automaton can be formally defined as follows:

Definition 1 *Hybrid automaton*

A hybrid automaton (HA) is a tuple $H=(L, V, B, E, \alpha, \eta)$ such that

- L is a set of locations
- V is a set of variables
- B is a set of synchronisation 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 labelling function that assigns to each location a set of *activities* which are time-invariant, and
- η is a labelling function that assigns to each location $l \in L$ an invariant condition $\eta(l) \subseteq \mathcal{V}$.

Linear hybrid systems are hybrid systems that have their activities, invariants, and transition relations all expressed as linear expressions on the system variables [4]. A *state* of a hybrid automaton H is a pair (l, v), where $l \in L$ and v is a

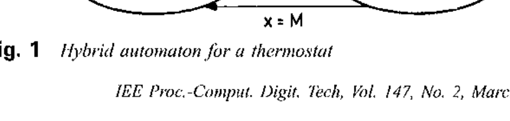

Fig. 1 *Hybrid automaton for a thermostat*

valuation of the variables in V. A *run* of H is a finite or infinite sequence

$$\rho : \sigma_0 \rightarrow^{t_0}_{f_0} \sigma_1 \rightarrow^{t_1}_{f_1} \ldots \tag{1}$$

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 is modelled as a network of linear hybrid automata (LHA). Two models are proposed: a *simple* model and a *network* model. In a simple model, 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. In a network model the hardware part is mapped into several LHA, each representing some physical hardware component, and the software part is also mapped into several LHA, each representing a software process.

In the following, the simple and network models are proposed for the two types of embedded system architectures found today, namely, 1-ASIC/1-CPU and multi-ASIC/multi-CPU, respectively. How the two models can also be interchangeably used is also explained.

3.1 Simple model

This model consists of only two LHA, one representing the hardware and one the software. This model is suitable for 1-ASIC/1-CPU embedded systems because there are only two clock rates: one for the ASIC and one for the CPU executing the software. If the interaction of the embedded system with its environment is to be verified then one more LHA is specified for modelling the environment. Synchronisation between the hardware and the software is achieved by declaring *synchronisation labels* on transitions of the LHA. A more complex system consisting of n-ASIC/m-CPU (n, $m > 0$) can also be modelled using the simple model, but the verification accuracy have to be traded-off for model simplicity and verification scalability because the clock rates are now declared as ranges instead of a single value. The rate range for hardware must cover all the n ASIC's clock rates and the rate range for the software must cover all the m CPU's clock rates. In this case, the simple model can only guarantee that if the hardware and the software clock rates are within that range, the system is correct or feasible, but it does not guarantee correctness or safeness for specific rate values. The network model as presented in the following is a more accurate one for such systems.

An example of a simple model is given in Fig. 2, which models a generic hardware-software system where the hardware waits for a specific period of time (h_{max}) for a response from the software. The hardware times out if the software does not respond within the time limit.

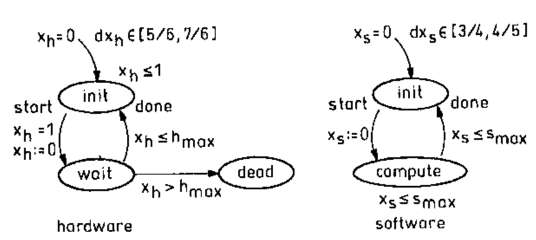

Fig. 2 *Maximum response-time synchronisation*

3.2 Network model

This model consists of more than two LHA, where either the hardware or the software or both are represented by collections of LHA. This model is suitable for an n-ASIC/m-CPU system, where each ASIC and each CPU can be represented by one LHA. In this case, there would be $n + m$ LHA in the network model. If the number of LHA is too large and thus affects verification, a more compact model can be obtained by modelling each type of ASIC and each type of CPU by a single LHA. Assuming that there is a small number of different types of ASICs and CPUs, the network model would be more manageable and verifiable. Although synchronisations among the hardware and software components can again be achieved by synchronisation labels as in the simple model, yet owing to the complex behaviour of the system, communication protocols are used instead. A communication protocol is modelled by an individual LHA, so as to maintain modularity and ease of verification. If two or more codesign alternatives, produced as a result of some codesign methodology, were similar in all respects, except for the communication protocol used in the hardware-software interface, then the network model can be easily reused by just replacing the LHA representing the communication protocol by a new one.

3.3 Parametric analysis and coverification

The design of embedded systems often involves parameters such as software execution time that depends on the processor executing it, hardware response time that depends on the technology and cost expended, software and hardware components costs that depend on the total system budget, and other system trade-offs. An intelligent choice of parameter values is a key decision in system design. *Parametric analysis* allows a system description (such as LHA or TA) to contain parameters and it finds a general solution in the form of a condition on parameter valuations for the system to satisfy given constraints or properties. For example, given a hardware-software system that must obey the *Fischer's mutual exclusion protocol* (FMEP) [4] and assuming the hardware process is executing at an execution rate 3.4 times faster than the software, parametric analysis finds the values for two parameters in the system. One parameter a represents the time required for lock-write, and another parameter b represents the time required for lock-read, where a, b are parameters, lock is a mutual exclusion variable, lock-write assigns a process index to lock, and lock-read checks if the value of lock is a process index. The conditions found by parametric analysis for this example are $5b \leq 17a$ and $a \geq 0$.

The following Section illustrates how through parametric analysis hardware-software systems can be coverified. Theoretically speaking, the coverification of the simple model and of the network model, presented in Sections 3.1 and 3.2, respectively, are both *decidable*. This is because both models are *simple multirate timed systems*. A simple multirate timed system is one that does not have any LHA comparing two or more skewed clock values, that is, clocks with different rates. It was proved in [4] (theorem 3.1) that the reachability problem for simple multirate timed systems is decidable. Since only reachability problems are considered in this article, timing coverification of the simple and network models is thus decidable.

3.4 HyTech tool

HyTech is a popular tool for verifying hybrid systems. Parallel composition of coordinating LHA is the basic system model. Parametric analysis can be performed by HyTech. *Regions* are data-structures used to represent a set of states symbolically through a set of linear constraints. HyTech uses polyhedra to represent convex zones and regions are composed of convex zones. There are two internal representations for polyhedra: set of linear constraints, and a *frame* consisting of points, rays, and lines acting as polyhedron generators. Time-step and transition-step successors are also computed symbolically in HyTech through manipulations of the polyhedra. HyTech computes the forward reachable region $post^*(I)$ by finding the limit of the infinite sequence $I, post(I), \ldots$ of regions, where $post(I)$ is defined as the set of states that are reachable, either by time-step or transition-step, from some state in I. Analogously, backward reachability can also be performed by finding the limit of predecessor regions.

Urgent and synchronised transitions can also be explicitly specified in the HyTech input language. In Section 4 synchronisation labels are used to synchronise transitions between hardware and software LHA.

4 Timing coverification techniques

Using the hybrid automata model for an embedded system, solution techniques are proposed for some commonly-found timing coverification problems. The five commonly-found elementary timing coverification problems presented here include: *maximum response-time* (MaxRT) synchronisation, *minimum reponse-time* (MinRT) synchronisation, *software concurrency verification* (SCV), *hardware concurrency verification* (HCV), and *integrated codesign-alternative verification* (ICAV). 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 Maximum response-time synchronisation

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 synchronisation 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 prespecified 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 maximum response-time synchronisations are successful for the given different time scales of the hardware and the software. In the following, the simple model is assumed (see Section 3.1) for ease of illustration and explanation.

4.1.1 MaxRT synchronisation problem: Given a hardware LHA $H=(L_H, V_H, B, E_H, \alpha_H, \eta_H)$ and a software LHA $S=(L_S, V_S, B, E_S, \alpha_S, \eta_S)$, the problem of verifying *maximum response-time* (MaxRT) synchronisation is defined as finding all conditions on the parameter variables in $V_H \cup V_S$ such that all hardware requests are responded by the software within the maximum wait-time of the hardware. A maximum wait-time is the largest period of time allowed for waiting.

4.1.2 Example: Fig. 2 shows an example of a simple LHA model for illustrating *maximum response-time* synchronisation. The hardware has a relative clock rate of [5/6, 7/6] and the software [3/4, 4/5]. The notation $[a, b]$ implies a closed interval, where a is the minimum rate and b is the maximum rate. Verifying the model using the HyTech tool, it was found that MaxRT synchronisation is guaranteed only if $9h_{max} \geq 14s_{max}$.

4.1.3 General solution: Analysis shows that if $[h_l, h_u]$ and $[s_l, s_u]$ were the hardware and software clock rates, respectively, the condition for MaxRT synchronisation can be given as a parametric expression

$$\bigwedge_{h_{max}, s_{max}} s_l h_{max} \geq h_u s_{max} \qquad (2)$$

where h_{max} is a maximum time the hardware, after making a request, will wait for the software response and s_{max} is a maximum time the software takes for computation of the requested task.

4.2 Minimum response-time synchronisation

In contrast to MaxRT synchronisation, *minimum response-time* (MinRT) synchronisation 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, 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.

4.2.1 MinRT Synchronisation problem: Given a hardware LHA $H=(L_H, V_H, B, E_H, \alpha_H, \eta_H)$ and a software LHA $S=(L_S, V_S, B, E_S, \alpha_S, \eta_S)$, the problem of verifying *minimum response-time* (MinRT) synchronisation is defined as finding all conditions on the parameter variables in $V_H \cup V_S$ such that all minimum wait-time constraints are satisfied by the hardware after making a request to the software. A minimum wait-time constraint is a restriction of waiting for a minimum period of time. □

4.2.2 Example: Fig. 3 shows an example of a simple LHA automata model for illustrating MinRT synchronisa-

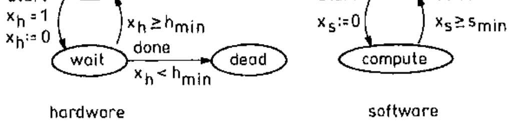

Fig. 3 *Minimum response-time synchronisation*

tion. 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: MinRT synchronisation is guaranteed when the parametric condition $25s_{min} \geq 24h_{min}$ is satisfied. □

4.2.3 General solution: Analytical study shows that if the hardware and software clock-rate ranges were $[h_l, h_u]$ and $[s_l, s_u]$, respectively, MinRT synchronisation is guaranteed only if the following condition is satisfied:

$$\bigwedge_{s_{min}, h_{min}} h_l s_{min} \geq s_u h_{min} \tag{3}$$

where s_{min} is a minimum computation time of the software and h_{min} is a 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 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 an m-processor system ($m \geq 1$) for ensuring system safety. The clock rates for each system configuration of the 1-processor, 2-processor, ..., m-processor ($m \geq 1$) must be estimated *a-priori*. The hardware waits for some maximum period of time after making a request for some software computation. By increasing the number of processors, the software performance could be improved and thus the software computation results could be produced within the hardware maximum wait-time.

4.3.1 Software concurrency coverification problem: Given a hardware LHA $H = (L_H, V_H, B, E_H, \alpha_H, \eta_H)$ and a software LHA $S = (L_S, V_S, B, E_S, \alpha_S, \eta_S)$, the problem of *software concurrency coverification* (SCC) is defined as finding an optimal software configuration, where S contains all the different configurations possible under the current cost limits. □

4.3.2 Example: Fig. 4 shows a hybrid automata model for a system with one hardware (H) and three possible software configurations: 1-processor (S_1), 2-processor (S_2) and 3-processor (S_3) 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 *sublinear* increase in computing power of the software configurations is assumed. If h_{max} is the maximum hardware wait-time and s_{max_i} is the maximum time required for software computation on an i-processor system, $i \in \{1, 2, 3\}$, on running through HyTech the following hold:

- (H, S_1) is safe if $3h_{max} \geq 8s_{max_1}$
- (H, S_2) is safe if $3h_{max} \geq 4s_{max_2}$ and
- (H, S_3) is safe if $9h_{max} \geq 8s_{max_3}$

In general, one can assume $s_{max_1} > s_{max_2} > s_{max_3}$ and hence form the following conclusions:

- all the three configurations are safe if $3h_{max} \geq 8s_{max_1}$
- only the 1-processor system is not safe if $3h_{max} \geq 4s_{max_2}$ and
- only the 3-processor system is safe if $9h_{max} \geq 8s_{max_3}$

Depending on the particular task at hand, h_{max} and s_{max_i} could be estimated and the degree of software concurrency (i.e. software configuration) obtained through software concurrency verification.

4.3.3 General solution: Given a hardware LHA H and a set of software LHAs $\{S_1, S_2, \ldots, S_k\}$, a configuration (H, S_i) is optimal if

$$\frac{h_{max}}{s_{max_i}} \geq \frac{h_u}{s_{il}} \quad \wedge \quad \frac{h_{max}}{s_{max_j}} < \frac{h_u}{s_{jl}}, \quad \text{for all } j < i \tag{4}$$

where the clock rate for hardware H is $[h_l, h_u]$ and the clock rate for software S_i is $[s_{il}, s_{iu}]$, for all $i > 0$. □

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.

4.4.1 Hardware concurrency coverification problem: Given a hardware LHA $H = (L_H, V_H, B, E_H, \alpha_H, \eta_H)$ and a software LHA $S = (L_S, V_S, B, E_S, \alpha_S, \eta_S)$, the problem of *hardware concurrency coverification* (HCC) is defined as finding an optimal hardware configuration,

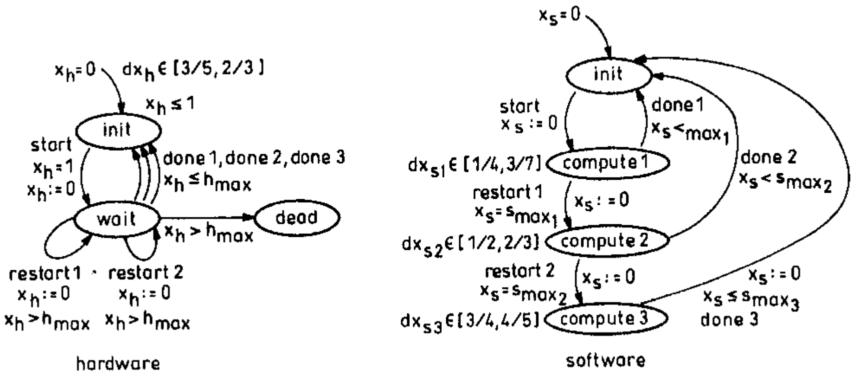

Fig. 4 *Software concurrency verification*

where H contains all the different configurations possible under the current cost limits. □

4.4.2 *Example:*
Fig. 5 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 [3/4, 4/5], [1/2,2/3], and [1/4, 3/7], 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 MinRT synchronisation coverification in Section 4.2) and s_{min} be the minimum computation time of software. Running this model through HyTech, the system configurations are safe only if the following conditions are satisfied:

- (H_1, S),(H_2, S), and (H_3, S) are safe if $9s_{min} \geq 8h_{min_1}$
- (H_2, S) and (H_3, S) are safe if $3s_{min} \geq 4h_{min_2}$
- Only (H_3, S) is safe if $3s_{min} \geq 8h_{min_3}$

Hence, if in the slowest and cheapest hardware configuration (H_3) the condition $3s_{min} \geq 8h_{min_3}$ is met one can use H_3 instead of the costlier H_1 and H_2 hardware configurations.

4.4.3 *General solution:*
Given a software LHA S and a set of hardware LHAs $\{H_1, H_2, \ldots, H_k\}$, a configuration ($H_i$, S) is optimal if

$$\frac{s_{min}}{h_{min_i}} \geq \frac{s_u}{h_{il}} \quad \wedge \quad \frac{s_{min}}{h_{min_j}} < \frac{s_u}{h_{jl}}, \quad \text{for all } j < i \qquad (5)$$

where the clock rate for hardware H_i is [h_{il}, h_{iu}], for all $i > 0$, and the clock rate for software S is [s_l, s_u].

In Sections 4.1–4.4, HyTech is a necessary tool for verification because, in general, an embedded system can be quite complex and must be modelled using the network model (Section 3.2). Simple analytic methods may be too tedious and error-prone.

4.5 *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 may depend on either the cost (minimum cost) or the performance (maximum throughput) or both (minimum cost-performance ratio). ICAV proposes a new criterion based on MaxRT and MinRT synchronisations, called *unsafe software-hardware rate-ratios* (USHeR). A ratio of software and hardware clock rates is called *safe* with respect to a particular system when all synchronisations (MaxRT and MinRT) in that system are satisfied. Two specific safe ratios are distinguished as *minimum safest* and *maximum safest*, corresponding to MaxRT synchronisations and MinRT synchronisations, respectively. The closed interval of minimum and maximum safest ratios is the *unsafe software-hardware rate-ratios* (USHeR), which will be the new criterion proposed for selecting an optimal codesign alternative in a design space exploration process of ICAV.

USHeR is called an *unsafe interval* because any software-hardware ratio values falling within this interval, not including the boundaries, will cause a violation of at least one synchronisation in the system. As special cases, when there are only MaxRT synchronisations, the upper bound of USHeR (i.e. maximum safest ratio) is taken as ∞ (infinity). Similarly, when there are only MinRT synchronisations, the lower bound of USHeR (i.e. minimum safest ratio) is taken as 0 (zero). When there are no synchronisations, the interval is undefined, which semantically indicates that any software-hardware ratio is allowed for a safe system.

Notationally, given a system with hardware clock-rate range [h_l, h_u] and software clock-rate range [s_l, s_u], USHeR is defined as the interval [h_l/s_u, h_u/s_l], where it is assumed that both MaxRT and MinRT synchronisations exist in the system. For any MaxRT sychronisation, $h_{max}/s_{max} \leq h_l/s_u$, where h_{max} and s_{max} are as defined in Section 4.1. For any MinRT synchronisation, $h_{min}/s_{min} \geq h_u/s_l$, where h_{min} and s_{min} are as defined in Section 4.2. 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 when the cost is especially low or the performance has peak bursts.

USHeR is best illustrated by an example as shown in Fig. 6. 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 MaxRT synchronisations. 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. Observe that under different metrics the best design configuration is different

- C_4 has the *least cost* but has very poor performance
- C_1 has the *best performance* but has very high cost
- C_2 has the *best cost–performance ratio*, but on applying ICAV to this example it was found to have the largest

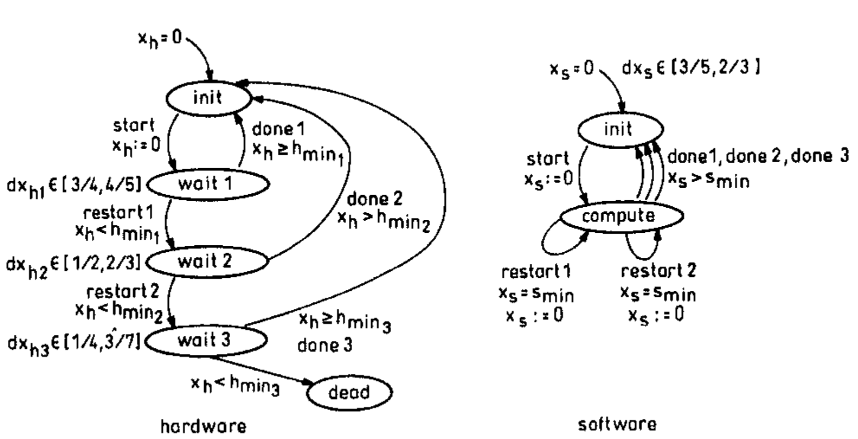

Fig. 5 *Hardware concurrency verification*

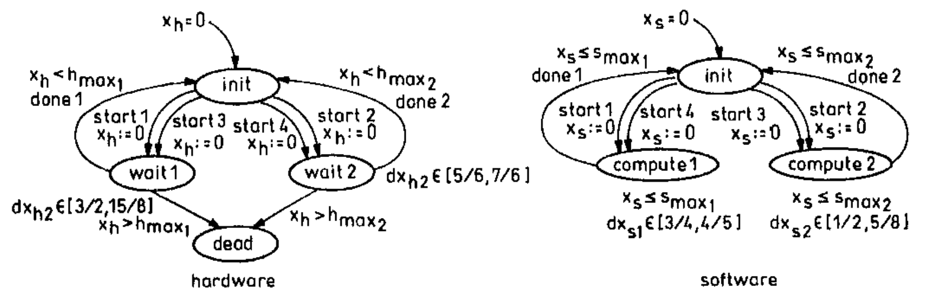

Fig. 6 *Integrated codesign alternative verification*

Table 1: ICAV example

Conf	HW Clock	SW Clock	Cost	Perf	Cost/Perf	USHeR	Safety Condition
C_1	[3/2, 15/8]	[3/4, 4/5]	1000	**100**	10.00	2.5	$2h_{max_1} \geq 5s_{max_1}$
C_2	[3/2, 15/8]	[1/2, 5/8]	750	80	**9.38**	3.75	$4h_{max_1} \geq 15s_{max_2}$
C_3	[5/6, 7/6]	[3/4, 4/5]	650	60	10.83	**1.56**	$9h_{max_2} \geq 14s_{max_1}$
C_4	[5/6, 7/6]	[1/2, 5/8]	**500**	50	10.00	2.33	$3h_{max_2} \geq 7s_{max_2}$

software-hardware incompatibility, that is the highest USHeR, which means synchronisation and other communications could require a large effort, and
• C_3 has the *least USHeR*, which means that the hardware and the software are the least incompatible and thus achieves a better hardware-software trade-off than the others.

4.6 Software-hardware-interface verification

A new modularised 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 synchronisation 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 [31] 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 works as follows. 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 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 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 transitions.
Briefly, the SHIV strategy is valid because if the global system is unsafe (some dangerous state is reachable), either the hardware or the software or the interface is faulty, but this leads to a contradiction because all three parts are verified safe by SHIV.

5 Ethernet bridge case study

Besides the five elementary problems presented in the previous Section, the approach has been applied to several real-world systems. An Ethernet bridge example is presented for illustration. Fig. 7 shows the process graph of an Ethernet bridge [10]. Bridges extend LANs by providing increased length, number of stations, performance, and reliability. They operate below the MAC service boundary, and is transparent to protocols operating above the boundary, in the logical link control (LLC). It is assumed as in [10] that the Ethernet LANs operate under the CSMA/CD access method. The basic function performed by bridges are: frame forwarding, learning station addresses, and resolving topology loops with the spanning tree algorithm. The communication estimates

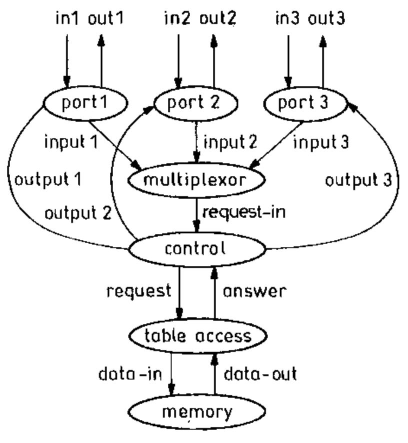

Fig. 7 *Ethernet bridge: process graph*

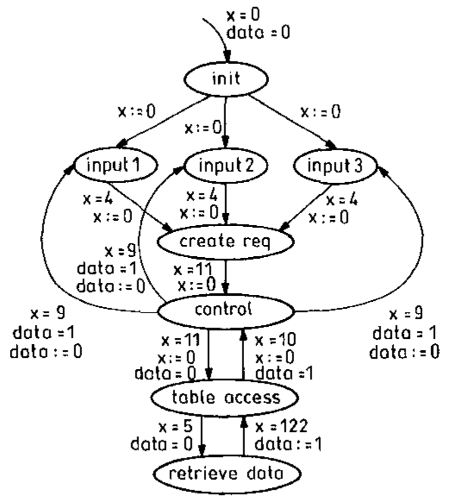

Fig. 8 *Ethernet bridge: linear hybrid automata model*

given in [10] were transformed into the linear hybrid automata model, which is illustrated in Fig. 8.

It was found that if the LHA model in Fig. 8 was directly verified using HyTech, it could not terminate even after modifying the system model as indicated in the HyTech user guide [6]. Here, the network model was adopted as described in Section 3.2 and finally, the SHIV strategy was applied. The decomposed hardware LHA and software LHA are shown in Fig. 9 and the interface LHA in Fig. 10. The interface verification is the most important for a codesign problem. The bridge processing rate was taken as 3000 pps (packets per second) and the hardware area constraint was assumed to be 4000 as in [10]. The hytech input code for interface verification is given in the Appendix (Section 8).

Observe that the Ethernet bridge example contains a typical MaxRT synchronisation problem. The software requires data from the memory and the hardware is responsible for accessing the memory and providing data to the software. The software waits for a maximum of s_{max} time units and the hardware returns data within a maximum of h_{max} time units.

Given a hardware clock range of [51/10, 6] and a software clock range of [1/5, 2/5], this example was modelled as a MaxRT synchronisation problem and input into the HyTech tool. The safety condition obtained was

$$4h_{max} \leq 51s_{max} \qquad (6)$$

For the estimates found in [10], h_{max} is 127 and s_{max} is 10, hence the equation is satisfied. This verifies that the bridge example is safe for the specific hardware and software clock ranges. Since eqn. 6 depends on the clock rates, an analytical study 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 as follows:

$$\frac{h_{max}}{s_{max}} \leq \frac{h_l}{s_u} \quad \text{or} \quad s_u h_{max} \leq h_l s_{max} \qquad (7)$$

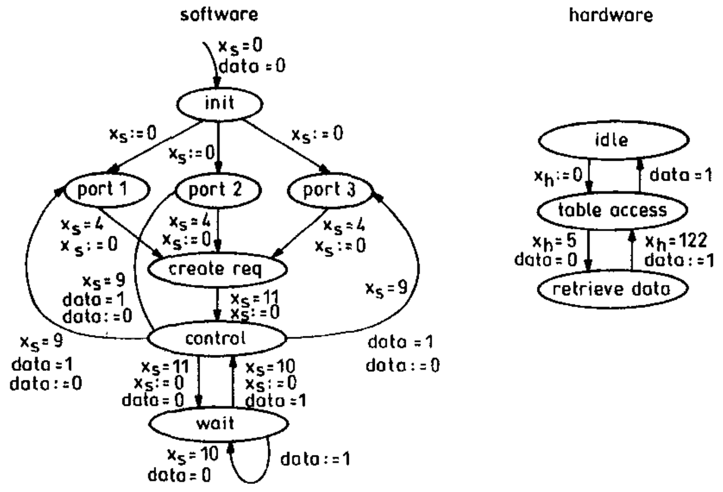

Fig. 9 *Ethernet bridge: hardware/software models*

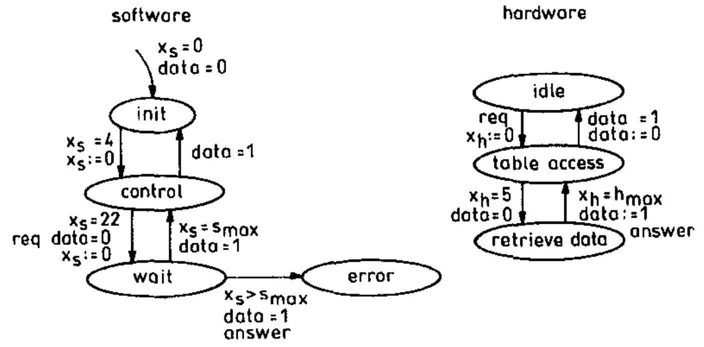

Fig. 10 *Ethernet bridge: interface models*

6 Conclusion

A linear hybrid automata model based timing coverification approach has been 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 new criterion for design selection based on coverification was also proposed. The new criterion called USHeR achieved a better tradeoff between hardware and software than conventional cost or performance-based metrics. 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.

7 References

1 ALUR, R., COURCOUBETIS, C., HALBWACHS, N., and DILL, D.: 'Model checking for real-time systems', Proceedings of IEEE Conference on *Logics in computer science*, Philadelphia, USA, 1990, pp. 414–425

2 HENZINGER, T.A., NICOLLIN, X., SIFAKIS, J., and YOVINE, S.: 'Symbolic model checking for real-time systems'. Proceeding of IEEE Conference on *Logics in computer science*, Santa Cruz, USA, 1992, pp. 394–406

3 ALUR, R., and DILL, D.: 'Automata for modeling real-time systems', *Theor. Comput. Sci*, 1994, **126**, (2), pp. 183–236

4 ALUR, R., COURCOUBETIS, C., HALBWACHS, N., HENZINGER, T.A., HO, P.-H., NICOLLIN, X., OLIVERO, A., SIFAKIS, J., and YOVINE, S.: 'The algorithmic analysis of hybrid systems', *Theor. Comput. Sci.*, 1995, **138**, (1), pp. 3–34

5 HOPCROFT, J.E., and ULLMAN, J.D.: 'Introduction to automata theory, languages, and computation' (Addison Wesley, 1979)

6 HENZINGER, T.A., HO, P.-H., and WONG-TOI, H.: 'A user guide to HyTech'. Proceedings of the LNCS conference on *Tools and algorithms for the construction and analysis of systems* (TACAS), 1995, Springer Verlag, **1019**, pp. 41–71

7 DAVEAU, J.M., MARCHIORO, G.F., BEN-ISMAIL, T., and JERRAYA, A.A.: 'COSMOS: An SDL based hardware/software codesign environment', *in* BERGE, J.-M., LEVIA, O., and ROUILLARD, J., (Eds.), 'Hardware/software co-design and co-verification' (Kluwer, 1997)

8 ANTONIAZZI, S., BALBONI, A., FORNACIARI, W., and SCIUTO, D.: 'The role of VHDL within TOSCA co-design framework'. Presented at Euro-VHDL, Grenoble, France, September 1994, pp. 612–617

9 AIGUIER, M., BENZAKKI, J., BERNOT, G., BEROFF, S., DUPONT, D., FREUND, L., ISRAEL, M., and ROUSSEAU, F.: 'ECOS: A generic codesign environment for the prototyping of real-time applications', *in* BERGE, J.-M., LEVIA, O., and ROUILLARD, J. (Eds.), 'Hardware/software co-design and co-verification' (Kluwer, 1997)

10 SANCHEZ, L., LOPEZ, M.L., MARTINEZ, N., CARRERAS, C., LOPEZ, J.C., DELGADO-KLOOS, C., ROYO, A., and BREUER, P.T.: 'Co-design at work: The ethernet bridge case study', *in* BERGE, J.-M., LEVIA, O., and ROUILLARD, J. (Eds.) 'Hardware/software co-design and co-verification' (Kluwer, 1997)

11 HSIUNG, P.-A.: 'CMAPS: A cosynthesis methodology for application-oriented parallel systems', *ACM Trans. Des. Autom. Electron. Syst.*, 2000, **5**, (1), pp. 51–81

12 GAJSKI, D., VAHID, F., and NARAYAN, S.: 'A system-design methodology: executable specification refinement'. Presented at European *Design automation and test* conference, February 1994, pp. 458–463

13 BALARIN, F., CHIODO, M., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., PASSERONE, C., SANGIOVANNI-VINCENTELLI, A., SENTOVICH, E., SUZUKI, K., and TABBARA, B.: 'Hardware-software co-design of embedded systems: the Polis approach' (Kluwer, 1997)

14 GUPTA, R.K., and DE MICHELI, G.: 'Hardware-software cosynthesis for digital systems', *IEEE Design Test Comput.*, 1993, **10**, (3), pp. 29–41

15 ERNST, R., HENKEL, J., and BENNER, T.: 'Hardware-software cosynthesis for micro-controllers', *IEEE Design Test Comput.*, 1993, **10**, (4), pp. 64–75

16 BUCHENRIEDER, K., and VEITH, C.: 'CODES: A practical concurrent design environment'. Presented at the international workshop on Hardware–Software Co-Design, 1992, pp. 12–13

17 CHOU, P.H., ORTEGA, R.B., and BORRIELLO, G.: 'The CHINOOK hardware-software co-synthesis system'. Proceeding of international symposium on *System synthesis*, September 1995, Cannes, France, pp. 22–27

18 HSIUNG, P.-A., LEE, T.-Y., and CHEN, S.-J.: 'MOBnet: An extended Petri net model for the concurrent object-oriented system-level synthesis of multiprocessor systems', *IEICE Trans. Inf. Syst.*, 1997, **E80-D**, (2), pp. 232–242

19 HSIUNG, P.-A., CHEN, S.-J., HU, T.-C., and WANG, S.-C.: 'PSM: An object-oriented synthesis approach to multiprocessor system design', *IEEE Trans. VLSI Syst.*, 1996, **4**, (1), pp. 83–97

20 HSIUNG, P.-A., CHEN, C.-H., LEE, T.-Y., and CHEN, S.-J.: 'ICOS: An intelligent concurrent object-oriented synthesis methodology for multiprocessor systems', *ACM Trans. Des. Autom. Electron. Syst.*, 1998, **3**, (2), pp. 109–135

21 HSIUNG, P.-A.: 'POSE: A parallel object-oriented synthesis environment', *ACM Trans. Des. Autom. Electron. Syst.*, **6**, (1), to appear January 2001

22 CHIODO, M., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., and SANGIOVANNI-VINCENTELLI, A.: 'Synthesis of software programs from CFSM specifications,' Technical report UCB/ERL M94/87, U.C. Berkeley, CA, 1994

23 CHIODO, M., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., and SANGIOVANNI-VINCENTELLI, A.: 'Synthesis of software programs from CFSM specifications'. Proceedings of *Design automation* conference, June 1995, San Francisco, USA, pp. 587–592

24 BALARIN, F., HSIEH, H., JURECSKA, A., LAVAGNO, L., and SANGIOVANNI-VINCENTELLI, A.: 'Formal verification of embedded systems based on CFSM networks'. Proceedings of *Design automation* conference, 1996, Las Vegas, USA, pp. 568–571

25 CHRISTOPHER, VIAL, and BRUNO, ROUZEYRE: 'Hardware-software co-synthesis: Modelling and synthesis of interfaces using interpreted petri nets', *in* BERGE, J.-M., LEVIA, O., and ROUILLARD, J. (Eds.), 'Hardware/software co-design and co-verification' (Kluwer, 1997)

26 BENGTSSON, J., LARSEN, K., LARSSON, F., PETTERSON, P., WANG, Y., and WEISE, C.: 'New generation of UPPAAL'. Presented at the international workshop on *Software tools for technology transfer* (STTT'98), July 1998, Aalborg, Denmark, pp. 43–51

27 HSIUNG, P.-A., and WANG, F.: 'A state-graph manipulator tool for real-time system specification and verification'. Presented at the 5th. IEEE international conference on *Real-time computing systems and applications* (RTCSA'98), October 1998, Hiroshima, Japan, pp. 181–188

28 WANG, F., and HSIUNG, P.-A.: 'Automatic verification on the large'. Proceedings of 3rd IEEE symposium on *High-assurance systems engineering* (HASE'98), November 1998, Washington DC, USA, pp. 134–141

29 HSIUNG, P.-A., and WANG, F.: 'User friendly verification'. Proceedings of the 1999 IFIP TC6/WG6.1 joint international conference on *Formal description techniques for distributed systems and communication protocols & protocol specification, testing, and verification* (FORTE/PSTV'99), October 1999, Beijing, China

30 HENZINGER, T.A., HO, P.-H., and WONG-TOI, H.: 'HyTech: The next generation'. Proceedings of the 16th IEEE symposium on *Real-time systems*, 1995, IEEE Computer Society Press, pp. 56–65

31 KUPFERMAN, O., and VARDI, M.Y.: 'On the complexity of branching modular model checking'. Proceedings of 6th LNCS international conference on *Concurrency theory*, **962**, August 1995