ELSEVIER

Journal of Systems Architecture 46 (2000) 1435–1450

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Embedded software verification in hardware–software codesign

Pao-Ann Hsiung *

Academia Sinica, Institute of Information Science, Academic Road, Sec. 2, No. 128 Taipei, Taiwan, Republic of China

Received 1 May 2000; accepted 1 September 2000

Abstract

Concurrent Embedded Real-Time Software (CERTS) is intrinsically different from traditional, sequential, independent, and temporally unconstrained software. The verification of software is more complex than hardware due to inherent flexibilities (dynamic behavior) that incur a multitude of possible system states. The verification of CERTS is all the more difficult due to its *concurrency* and *embeddedness*. The work presented here shows how the complexity of CERTS verification can be reduced significantly through answering common engineering questions such as *when*, *where*, and *how* one must verify embedded software. First, a new *Schedule-Verify-Map* strategy is proposed to answer the *when* question. Second, verification under *system concurrency* is proposed to answer the *where* question. Finally, a complete *symbolic model checking* procedure is proposed for CERTS verification. Several application examples illustrate the usefulness of our technique in increasing verification scalability.

Keywords: Embedded software; Software verification; Symbolic model checking; System/process concurrency; Quasi-static scheduling; Software synthesis

1. Introduction

With the burgeoning wide-spread embedding of software into computerized systems and the increasing complexity of today's hardware–software systems, *software verification* is an indispensable procedure in system synthesis. Software verification tries to uncover discrepancies in the interaction between software and hardware, to guarantee the satisfaction of real-time constraints by the software under all circumstances, and to check if the synthesized software is optimally configured. In this work, we try to answer questions related to software verification such as *when* should software be verified, *where* should software be verified, and *how* should software be verified.

When should software be verified? Embedded software is synthesized through a process called *quasi-static scheduling* (QSS) [25]. QSS computes most of the schedule for a set of software processes at compile time, leaving at run-time only the solution of data-dependent decisions. After QSS, software code is then generated through a simple syntax mapping from the scheduled processes. Verification can be performed at three different points: before scheduling, after scheduling, and after code generation. On one hand, before scheduling, processes generally have some regions in its state-space which will be eventually eliminated by scheduling. On the other hand, after code

* Tel.: +886-2-2788-3799; fax: +886-2-2782-4814.
E-mail address: hpa@computer.org (P.-A. Hsiung).

1383-7621/00/$ - see front matter
PII: S1383-7621(00)00034-5

generation, the software code is generally implementation-dependent and contains coding technicalities that do not really contribute toward the actual behavior of the software. Hence, we propose to verify software *after scheduling* and *before code generation*, which will be discussed in details in Section 3.

Where should software be verified? There are generally two kinds of concurrencies in a hardware–software system: *system* concurrency and *process* concurrency. System concurrency is generally associated with the number of *Central Processing Units* (CPUs) or *Application Specific Instruction Processors* (ASIPs) or *Application Specific Integrated Circuits* (ASICs). Normally there are dedicated CPUs for software execution. Thus, we will confine ourselves to the number of CPUs executing the software as the *system concurrency*. Process concurrency is the maximum degree of parallelism that a set of processes exhibits during execution. Process concurrency is generally much larger than system concurrency. This is because if process concurrency were to be smaller than system concurrency then the system will be under-utilized, indicating a waste of resources. Conventionally, software is verified under process concurrency. For example, we usually verify a communication protocol under the maximum-process concurrency assumption. What we propose here is that embedded software should instead be verified under the system concurrency. This has a significant impact on verification efficiency and scalability, as will be illustrated in Section 4.

How should software be verified? An algorithmic procedure for formal verification that has gained unforeseen popularity among verification scientists and likewise among design engineers, is called *model checking*. Model checking is an automatic procedure to verify if a given system satisfies a given temporal property [4]. For dense real-time systems, a system is often described using *Timed Automata* (TA) [7] and a property is often specified in *Timed Computation Tree Logic* (TCTL) [14,15]. For hybrid systems, *Linear Hybrid Automata* (LHA) [5,6] is a theoretically proven model for verification purposes. LHA was also recently used as a formal model for timing coverification of hardware/software systems by the author [16]. Here, we assume that all processors in our target system are homogeneous. Since we are mainly verifying concurrent software that executes on processors with the same clock rates, we will be using the simpler TA model instead of LHA. We propose two model-checking algorithms for *Concurrent Embedded Real-Time Software* (CERTS). Basically, the algorithms work by abridging a set of given TA into a smaller set to acquiesce for the smaller system concurrency (as compared to process concurrency) and then annotating the abridged TA with pre-generated valid schedules. Finally, model checking is applied on the abridged and annotated set of TA. A detailed description is given in Section 5.

This paper is organized as follows. Section 2 gives a brief survey of current software synthesis methods in context of verification. Section 3 answers the *when* question by proposing a *Schedule-Verify-Map* (SVM) strategy. Section 4 answers the *where* question by demonstrating the validity of verifying at the *system concurrency* instead of at process concurrency. Section 5 answers the *how* question by proposing two verification algorithms for CERTS. Section 6 gives several application examples to support the answers presented in the previous three sections. Section 7 concludes the paper.

2. Embedded software synthesis

Currently, *software synthesis* is a hot topic of research in the field of hardware–software codesign. Previously, a large effort was directed towards hardware synthesis and comparatively little attention paid to software synthesis. Partial software synthesis was mainly carried out for communication protocols [24], plant controllers, [8,9,23] and real-time schedulers [2,26] because they generally exhibited regular behaviors. Only recently has there been some work on automatically generating software code for embedded systems [10,21,22,25,27]. As far as the authors know, no automatic software synthesis method is available for concurrent real-time embedded software. We are working on this portion of research,

and this paper is the verification part of the work. In the following, we will briefly survey the existing works on the synthesis of non-real-time software.

Lin [21,22] proposed an algorithm that generates a software program from a concurrent process specification through intermediate Petri-Net representation. This approach is based on the assumption that the Petri-Nets are safe, i.e., buffers can store at most one data unit, which implies that it is always schedulable. The proposed method applies QSS to a set of safe Petri-Nets to produce a set of corresponding state machines, which can then be mapped syntactically to the final software code. Zhu and Lin [27] proposed a compositional synthesis method that reduced the generated code size and thus was more efficient.

A software synthesis method was proposed for a more general Petri-Net framework by Sgroi et al. [25]. A QSS algorithm was proposed for *Free-Choice Petri Nets* (FCPN) [25]. A necessary and sufficient condition was given for a FCPN to be schedulable. Schedulability was first tested for a FCPN and then a valid schedule generated by decomposing a FCPN into a set of *Conflict-Free* (CF) components which were then individually and statically scheduled. Code was finally generated from the valid schedule.

Balarin et al. [10] proposed a software synthesis procedure for reactive embedded systems in the *Codesign Finite State Machine* (CFSM) [11] framework with the POLIS hardware–software codesign tool [11]. This work cannot be easily extended to other more general frameworks.

All the above work suggests that the research on software synthesis is still at a very young stage and *without any verification*. We propose to incorporate software verification into the synthesis procedure for several reasons. Firstly, with the increased use of *Virtual Components* (VC) or *Intellectual Properties* (IP), an embedded software might be installed into more than one type of system, hence its adaptability to various system environments should be verified. Secondly, since software is inherently more complex than hardware, mere simulation or testing might not uncover errors that could be drastic. Hence, verification is required especially for high assurance systems. Thirdly, certain features of the software part in a hardware–software system are independent of the hardware. Verifying these features together with the hardware would unnecessarily increase verification complexity. Thus, isolating software-specific faults might increase the chances of successfully coverifying the entire system. Lastly, if verification is performed after the complete software synthesis procedure, then the implementation details in the generated software code would simply obscure important behaviors of the software, thus making software more difficult to verify.

In the following, we will illustrate our verification approach based on the QSS method of software synthesis. Our verification approach is independent of the scheduling method and hence can be applied also to other methods of software synthesis such as priority-based preemptive scheduling.

3. Schedule-verify-map strategy

This section answers the *when* question, that is, "*when should software be verified*?" The context in which software verification will be discussed is *embedded software synthesis*, which was described in Section 2. As depicted in Fig. 1, there are three stages in synthesis (first row in the figure), namely, *process specification*, *scheduling*, and *code generation*.

- *Stage* (1). In process specification, a set of communicating processes representing the behavior of desired software is specified. This specification can be in the form of a set of Petri-Nets [21,22,25,27], or in formal specification languages such as Esterel, LOTOS, and others.
- *Stage* (2). In scheduling, except for run-time dependent computations, all other computations in the specified processes are quasi-statically scheduled [22,25]. The scheduled processes are usually represented by a set of finite state-machines.
- *Stage* (3). In code generation, the set of finite state-machines is syntactically mapped to actual software code. A software time loop may be utilized to maintain the schedule in the finite state-machines.

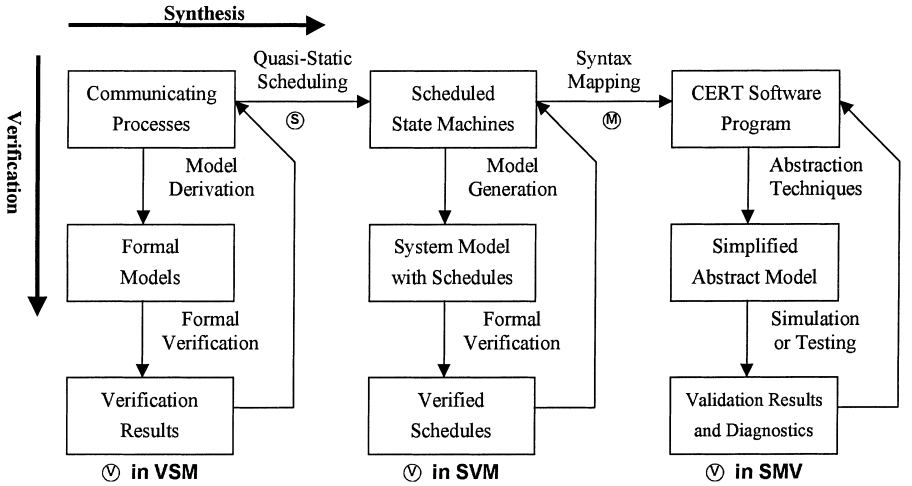


Fig. 1. En-route verification for concurrent embedded real-time software synthesis.

3.1. Conventional verification approaches

Theoretically, verifying the given processes can be done after either one of the stages during software synthesis. Verification scientists try to verify processes immediately after process specification (i.e., Stage (1)) to find any specification errors. This is called the *Verify-Schedule-Map* (VSM) approach (column 1 and row 1 in Fig. 1). Design engineers try to verify the final program after code generation (i.e., Stage (3)). This is called the *Schedule-Map-Verify* (SMV) approach (row 1 and column 3 in Fig. 1). Both of these approaches encounter different degrees of state-space explosion problems.

Verifying process specification explores unnecessary regions in the state-space that would eventually not even exist in the final software code. These regions are basically those that will be eliminated after scheduling (Stage (2)). The problem becomes worse when the degree of non-determinism is high in the specification or when the degree of process concurrency increases. The *degree of non-determinism* (δ_{ND}) is the maximum number of different possible behaviors that a system can have in any one state. The *degree of process concurrency* (δ_{PC}) is defined as the maximum number of processes that can execute concurrently in the system specified. Further details on how δ_{ND} and δ_{PC} affect verification are discussed in Sections 3.2 and 4.

Verification of software program code also indulges in unnecessary state-space explosions and thus affects scalability in the number or size of processes verifiable. Software programs usually contain many auxiliary, implementation-dependent variables, that contribute towards neither the real behavior of the software nor the satisfaction of specified real-time constraints by the software. As is well-known, the state-space size explored during verification increases exponentially with the number of clock variables and largest integer constant used [4]. The state-space size also increases drastically with the number of free variables. Software programs generally contain a lot of variables, the number of which is not optimized either by the software synthesis procedure or by the software compiler.

In conclusion, both of the above approaches unnecessarily explore regions in the state-space that do not contribute towards the actual goal of verification. Thus, in Section 3.2 a new approach is proposed called SVM as illustrated by row 1 and column 2 of Fig. 1.

3.2. Proposed SVM approach

To overcome the difficulties in verification presented in Section 3.1, we propose a new approach called SVM. In SVM, verification is performed

Table 1
Comparison of three verification approaches

	Verification approach	Correctness	Feasibility	State-space size	Completeness
(1)	VSM	Too sure	Vague	Exponentially large	More than complete
(2)	SVM	Sure	Largely	Reduced	Complete
(3)	SMV	Not sure	Practical	Small to Medium	Incomplete

after scheduling and before code generation. Since scheduling eliminates certain regions in the state-space, SVM will obviously explore a much smaller part of the state-space. The degree of reduction is analyzed in Section 3.3. Since the target of verification is a set of scheduled processes and *not* program code, SVM will also search a smaller state-space than the engineers' approach (verification after code generation).

Comparing the two conventional approaches – VSM adopted by verification scientists, SMV adopted by design engineers, and our proposed SVM approach, we have the pros and cons of each summarized in Table 1. On comparison, it is observed that SVM is a good trade-off between practical feasibility (column 4) and verification completeness (column 6). Although VSM is more than complete, its practical feasibility is often hindered by the exponentially large state-space. SMV is the most practical among the three approaches, yet it is an incomplete validation process (mostly accomplished through simulation and testing that covers less than 100% of the system behavior). A more detailed analysis on the SVM approach is presented in Section 3.3.

3.3. Analysis

To analyze the advantage of the SVM strategy in comparison with the conventional approaches, some formal notations, definitions, and results are necessary. The sets of integers and non-negative real numbers are denoted by $\mathcal{N}$ and $\mathcal{R}_{\geqslant 0}$, respectively.

TA is composed of various *modes* interconnected by *transitions*. Variables are segregated into categories of *clock* and *discrete*. Clock variables increment at a uniform rate and can be reset on a transition, whereas discrete variables change values only when assigned a new value on a transition. A TA may remain in a particular mode as long as the values of all its variables satisfy a *mode predicate*, which is a conjunction of clock constraints and boolean propositions.

Definition 1 (*Mode predicate*). Given a set C of clock variables and a set D of discrete variables, the syntax of a *mode predicate* η over C and D is defined as: $\eta := \mathit{false} \mid x \sim c \mid x - y \sim c \mid \ d \sim c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where $x,y \in C$, $\sim \in \{\leqslant, <\ , =, \geqslant, >\}$, $c \in \mathcal{N}$, $d \in D$, and η_1, η_2 are mode predicates. □

Let $B(C,D)$ be the set of all mode predicates over C and D. A TA may go from a mode to another, that is perform a transition, when the triggering condition (also specified as a mode predicate) is satisfied by the current valuation of clock and discrete variables. On a transition, some clocks may be reset to zero and some discrete variables may be assigned new integer values.

Definition 2 (*Timed automaton*). A timed automaton (TA) is a tuple $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$ such that: M_i is a finite set of modes, $m_i^0 \in M$ is the initial mode, C_i is a set of clock variables, D_i is a set of discrete variables, $\chi_i : M_i \mapsto B(C_i, D_i)$ is an *invariance* function that labels each mode with a condition true in that mode, $E_i \subseteq M_i \times M_i$ is a set of transitions, $\tau_i : E_i \mapsto B(C_i, D_i)$ defines the transition triggering conditions, and $\rho_i : E_i \mapsto 2^{C_i \cup (D_i \times \mathcal{N})}$ is an *assignment* function that maps each transition to a set of assignments such as resetting some clock variables and setting some discrete variables to specific integer values. □

Definition 3 (*System state*). Given a system $\mathcal{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modeled by a set of n TA, $\{\mathcal{A}_i \mid \mathcal{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i), 1 \leqslant i \leqslant n\}$, a

state s of system $\mathscr{S}$ is defined as a mapping from $\{1,\ldots,n\} \cup \bigcup_i C_i \cup \bigcup_i D_i$ to $\bigcup_{1\leqslant i\leqslant n} M_i \cup \mathscr{N} \cup R_{\geqslant 0}$ such that

- $\forall i \in \{1,\ldots,n\}$, $s(i) \in M_i$ is the mode of $\mathscr{A}_i$ in s,
- $\forall i, \forall x \in C_i, s(x) \in R_{\geqslant 0}$ is the reading of clock x in s, such that $s(x) \models \bigwedge_i \chi_i(s(i))$, and
- $\forall i, \forall d \in D_i, s(d) \in \mathscr{N}$ is the value of d in s, such that $s(d) \models \bigwedge_i \chi_i(s(i))$. □

Definition 4 (*Mode transition*). Given a system $\mathscr{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modeled by a set of n TA, $\{\mathscr{A}_i \mid \mathscr{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i),\ 1 \leqslant i \leqslant n\}$, and two system states s and s', there is a *mode transition* from s to s' in $\mathscr{S}$, in symbols $s \to s'$, iff there is an $1 \leqslant i \leqslant n$ such that

- $(s(i), s'(i)) \in E_i$;
- $s(i) \models \tau_i(s(i), s'(i))$;
- for all $1 \leqslant j \leqslant n$ and $j \neq i$, $s(j) = s'(j)$;
- $\forall x \in X((x \in \rho_i(s(i), s'(i)) \Rightarrow zs'(x) = 0) \wedge (x \notin \rho_i(s(i), s'(i)) \Rightarrow s'(x) = s(x)))$. □

With the above given definitions on TA, system state, and mode transition, we will start analyzing SVM by first defining the concepts of *state non-determinism* in a system, and then using it to quantify the benefits obtain by SVM.

Definition 5 (*State non-determinism*). Given a system $\mathscr{S}$ of n processes $\{P_1, P_2, \ldots, P_n\}$ modeled by a set of n TA, $\{\mathscr{A}_i \mid \mathscr{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i), 1 \leqslant i \leqslant n\}$, and a state s of system $\mathscr{S}$, the *state non-determinism* of s is defined as the total number of mode transitions ($s \to s'$), whose occurrence at s is non-deterministic (arbitrarily decided), where s' is a successor system state. Notationally, we have the following definition for the state non-determinism ($\psi(s)$) at s:

$$\psi(s) = \Pi_{1\leqslant i\leqslant n} \omega(s(i)), \tag{1}$$

where $\omega(m)$ is the number of out-going transitions of a mode m, which are non-deterministic. □

In general, not all state non-determinism ($\psi(s)$) at a state (s) can be quasi-statically scheduled. We denote by $\psi_{\text{qss}}(s)$ those non-determinism that *can be quasi-statically scheduled*. In notations,

$$\psi_{\text{qss}}(s) = \Pi_{1\leqslant i\leqslant n} \omega_{\text{qss}}(s(i)), \tag{2}$$

where $\omega_{\text{qss}}(m)$ is the number of out-going transitions of a mode m, which are non-deterministic and can be quasi-statically scheduled.

Considering the overall effect of QSS on verification complexity, we have the following results. First, given a state s of a system $\mathscr{S}$, since all QSS non-determinism have been eliminated before SVM, the reduction obtained is a multiplicative factor of $\psi_{\text{qss}}(s)$ for a state s. Second, along a computation run of a system (that is, a sequence of alternating states and mode transitions), the combined effect of state non-determinisms at a state s and a successor state s' is multiplicative. This means that the combined non-determinism is $\psi_{\text{qss}}(s) \times \psi_{\text{qss}}(s')$. Thus, the overall reduction effect of QSS on a system behavior can be quantified by the total number of non-determinisms, $\Psi_{\text{qss}}(\mathscr{S})$, resolved by QSS for a system $\mathscr{S}$, as follows.

$$\begin{aligned}\Psi_{\text{qss}}(\mathscr{S}) &= \prod_{s\in \mathbf{Reach}(\mathscr{S})} \psi_{\text{qss}}(s) \\ &= \prod_{s\in \mathbf{Reach}(\mathscr{S})} \prod_{1\leqslant i\leqslant n} \omega_{\text{qss}}(s(i)),\end{aligned} \tag{3}$$

where $\mathbf{Reach}(\mathscr{S})$ is the set of reachable states of system $\mathscr{S}$.

Here, the *resolution* of a set of non-determinisms at a state, s, means instead of considering all possible successor states, s', due to the concurrent non-determinisms in each process, QSS has *fixed* (that is, scheduled) *only one* of the successor states as a valid scheduled state, where $s \to s'$. Hence, the total number of computation runs that SVM explores is $\Psi_{\text{qss}}(\mathscr{S})$ times less than that explored by the VSM approach for a system $\mathscr{S}$.

Taking limits on $\Psi_{\text{qss}}(\mathscr{S})$, we find that it is a *double exponential* term in the number of system processes, n, and in the size of the reachable state-space $|\mathbf{Reach}(\mathscr{S})|$, as given in the following.

$$\Psi_{\text{qss}}(\mathscr{S}) \to (\delta_{\text{ND}})^{n\times|\mathbf{Reach}(\mathscr{S})|}, \tag{4}$$

where δ_{ND} is the maximum degree of non-determinism of all processes, $p_1, p_2, \ldots, p_n$. This shows a double exponential decrease in the number of computation runs that need be explored by SVM compared to VSM.

The above was an analytical comparison between the proposed SVM approach and the VSM approach. For a comparison between SVM and SMV, since each final generated program code might contain different number of auxiliary variables and data structures, it is difficult to analyze theoretically. Nevertheless, the number of computation runs explored by SMV will be definitely larger than that by the SVM approach due to an increase in state-space size with an increase in the number of variables.

4. Handling concurrency

This section answers the *where* question, that is, "*where should software be verified*?" The context in which software verification will be discussed is *embedded software synthesis*, which was described in Section 2. In general, embedded software is executed on one or more embedded microprocessors. The *system concurrency*, δ_{SC}, as defined in Section 1, is the number of CPUs allocated for executing an embedded software program. The *degree of process concurrency*, δ_{PC}, as defined in Section 1, is the maximum number of processes that can execute concurrently in the system specified. As described in the rest of this section, we propose to verify embedded software under system concurrency, rather than under process concurrency.

4.1. Process concurrency and system concurrency

Nearly all verification theory is based on process concurrency. In general, a system is specified as consisting of a set of processes and the system is then expected to be verified under the assumption that all the processes execute concurrently. However, in the real-world of computerized embedded systems, concurrency is generally limited by the embedding system, that is, the hardware architecture that provides an execution environment. The former is process concurrency, while the latter is system concurrency. System concurrency is generally much smaller than process concurrency. For example, in a two-processor system executing 200 processes concurrently, the process concurrency is 200, but the system concurrency is only two.

Concurrent software is also generally specified as a set of processes. For example, an image processing task may consists of several concurrent processes working on a part of the image data, but if there are only two processors then there can be at most only two processes working concurrently. Communication protocol algorithms are also considered to be executed by a set of processes running concurrently. For example, a system of 10 processes, obeying the *Fischer's Mutual Exclusion Protocol* (FMEP) [1,19,20] and executing on five processors, has a maximum degree of concurrency of five, and not 10. TA for a typical process obeying FMEP is given in Fig. 2.

4.2. Verification under different concurrencies

The scalability of formal verification, especially that of model checking, strictly depends on inherent concurrencies in a system model. The size of state-spaces explored by model checking grows exponentially with an increase in concurrency. For example, a two-process system obeying FMEP has 70 modes and 160 transitions [18], a three-process system has 1239 modes and 4013 transitions, and a 4-process system has approximately 28K modes and 120K transitions. The increase is drastic.

The concurrency of a system is generally specified as the number of processes running in the system. This is incorrect when embedded systems are concerned, because the actual concurrency (number of processors) is much smaller than the

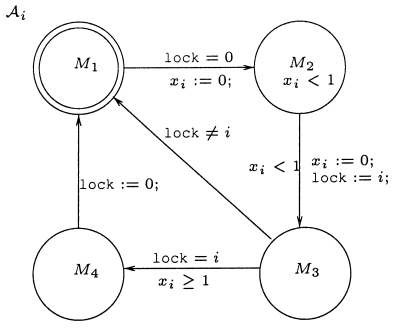


Fig. 2. Fischer's mutual exclusion protocol (*i*th process).

number of processes. For example, if verification is performed for a four-process *signal polling system* executing on two processors, then the size of state-space explored is only 57 modes and 79 transitions, which is much smaller than that for a 4-process system verified under process concurrency of four (78K modes and 205K transitions). TA for the signal polling system is given in Fig. 3.

For ease of discussion, we will adopt the following notations. In the rest of this paper, unless mentioned otherwise, assume we are given a system $\mathscr{S}$ with n processes $\mathscr{P} = \{P_1, P_2, \ldots, P_n\}$, modeled by n TA $\mathscr{A} = \{\mathscr{A}_1, \mathscr{A}_2, \ldots, \mathscr{A}_n\}$, respectively, where $\mathscr{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$, $1 \leqslant i \leqslant n$. Also, assume there are m processors in system $\mathscr{S}$, that is, $\mathscr{Q} = \{Q_1, Q_2, \ldots, Q_m\}$. Hence, a system is defined as a two-tuple $\mathscr{S} = \langle \mathscr{P}, \mathscr{Q} \rangle$.

On software synthesis, the n processes in $\mathscr{P}$ are QSS on the m processors in $\mathscr{Q}$ (refer to Section 2 for software synthesis methods). Let Z be the set of valid schedules generated by any software synthesis method, that is, $Z = \{\zeta_i \mid \zeta_i = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_r} \rangle, P_{k_j} \in \mathscr{P}, 1 \leqslant j \leqslant r \leqslant n, 1 \leqslant i \leqslant m\}$, where $\zeta_i = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_r} \rangle$ is a schedule for processor Q_i, such that processes $P_{k_1}, P_{k_2}, \ldots, P_{k_r}$ are scheduled to run on Q_i. Here, it is assumed that processes are scheduled non-preemptively, which is not a severe restriction as most preemptions can always be removed by breaking a process into two or more at its preemption points. Further, *loop repetitions* in a schedule can be denoted by a bar on a sequence of processes with an integer on top of the bar to denote the number of times the loop is repeated. For example,

$$\langle P_1, \overline{P_3, P_5}^{\,3}, P_8, P_{10} \rangle$$

is a schedule for first executing P_1, then executing three times the sequence of P_3, P_5, and finally the rest of the schedule P_8, P_{10}.

The main issue in handling concurrency is how do we verify n processes under the system concurrency of m processors. In the following Sections 4.3 and 4.4, we propose two different approaches for solve this issue, namely, *processor-oriented verification* and *process-oriented verification*.

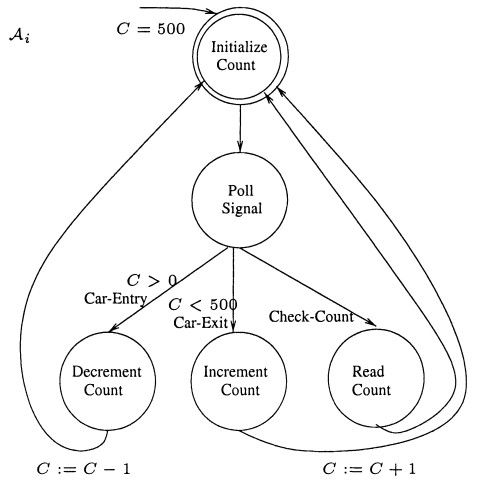


Fig. 3. Signal polling system.

4.3. Processor-oriented verification

A straightforward method is to create a new TA for modeling the behavior of each processor. This is called *processor-oriented verification*. Besides being straightforward, it can be easily extended to include process preemptions due to the flexibility in directly incorporating schedule changes into the structure of a processor timed automaton.

Based on the syntax representation of a TA, we know that each process automaton, $\mathscr{A}_i$, either has a transition, $e_f \in E_i$, that loops back to the initial mode, m_i^0, from some mode in $M_i \setminus \{m_i^0\}$ or has a final mode, $m_f \in M_i$. A *looping transition* is defined as one that loops back to the initial mode from some non-initial mode. A *final mode* is defined as an accepting mode, from which there is no outgoing transition.

A processor timed automaton, $\mathscr{B}_i = (M_i', m_i'^0, C_i', D_i', \chi_i', E_i', \tau_i', \rho_i')$, is constructed for processor Q_i as follows. For each process, P_{k_j}, that appears in the schedule ζ_i, include the process automaton $\mathscr{A}_{k_j}$ into $\mathscr{B}_i$. The inclusion method involves how two consecutive TA, $\mathscr{A}_{k_j}$ and $\mathscr{A}_{k_{j+1}}$ are to be merged into the new $\mathscr{B}_i$. For each looping transition, e_f in E_{k_j}, change the destination mode of e_f into the initial mode, $m_{k_{j+1}}^0$, of $\mathscr{A}_{k_{j+1}}$. For each final mode, m_f, create a new transition, e_f', from m_f to the initial mode, $m_{k_{j+1}}^0$, of $\mathscr{A}_{k_{j+1}}$. Thus, transitions e_f and e_f' interconnect the two consecutive TA, $\mathscr{A}_{k_j}$

and $\mathscr{A}_{k_{j+1}}$ in the new TA $\mathscr{B}_i$. Recall that a loop may exist in a schedule, as denoted by an overline bar and a number indicating the number of times the loop must execute. Suppose a partial schedule

$$\overline{P_{k_j}, \ldots, P_{k_{j+v}}}^{\,u}$$

is to be looped for u times, where $u > 1$, $v \geqslant 0$, and $k_j, \ldots, k_{j+v} \in \{1, \ldots, n\}$. Counter variables are created to keep count of the number of times the loop has executed. Interconnecting transitions connect $\mathscr{A}_{k_{j+v}}$ with the initial mode of $\mathscr{A}_{k_v}$ and with the initial mode of the next process after the loop in a schedule.

The set of newly created processor automata, $\mathscr{B} = \{\mathscr{B}_1, \ldots, \mathscr{B}_m\}$, is then used for the system model in the model-checking procedure as described in Section 5.

4.4. Process-oriented verification

Another method of verifying n processes, running under the system concurrency of m processors, is by directly restricting the execution of the process timed automata in $\mathscr{A}$. This approach is called *process-oriented verification*. This approach does not allow process preemption, but is more elegant.

A *processor locking variable* is used to restrict the execution of a process according to a schedule. A processor locking variable is a mutual exclusion variable that indicates which process is currently being executed on a processor. For example, a processor locking variable, l_k, locks processor Q_k and if $l_k = k_j$, then process P_{k_j} is currently being executed on processor Q_k.

Modifications of process timed automata are carried out as follows. Create a set of m processor locking variables, $\{l_1, \ldots, l_m\}$, such that l_k locks processor Q_k, $1 \leqslant k \leqslant m$. Suppose the processor schedules are as follows: $\zeta_k = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_r} \rangle$, $1 \leqslant k \leqslant m$, $1 \leqslant r \leqslant n$. Let the initial value of l_k be k_1. Assume that process P_{k_j} is scheduled on processor Q_k, $1 \leqslant j \leqslant r$. Modify each process timed automaton, $\mathscr{A}_{k_j}$, as follows:

- Create a new initial mode, $m'^0_{k_j}$, for $\mathscr{A}_{k_j}$,
- Create a new transition, e^0, leading from $m'^0_{k_j}$ to the original initial mode $m^0_{k_j}$,
- Let the triggering condition $\tau_{k_j}(e^0)$ be $l_k = k_j$.
- For each looping transition (defined in Section 4.3), e, let $\rho_{k_j}(e) = \rho_{k_j}(e); (l_k := k_{j+1})$, where $P_{k_{j+1}}$ is the next process scheduled to be executed on processor Q_k after P_{k_j} and ";" is a concatenation operator for an assignment statement.
- For each loop repetition in a schedule (denoted by an overline bar), a counter variable is created to count the number of times the loop has executed.

A basic assumption made here is that each process is scheduled on a single processor and no preemption is allowed. The set of *modified* process timed automata in $\mathscr{A}$ is now denoted by $\mathscr{A}' = \{\mathscr{A}'_1, \ldots, \mathscr{A}'_n\}$, which is finally input to the model-checking procedure as described in Section 5.

5. Model checking embedded software

The framework of verification that we use for software verification is the popular *model checking* framework [4,15], as introduced in Section 1. Model checking verifies if a given system satisfies a given property. In our framework, a real-time system is described using TA [7] (see Definition 2) and a temporal property is specified using *Timed Computation Tree Logic* (TCTL) [14,15] (see Definition 6).

In our framework, recall from Section 4.2, a system $\mathscr{S} = \langle \mathscr{P}, \mathscr{Q} \rangle$ is composed of the following components:

- a set of n processes, $\mathscr{P} = \{P_1, \ldots, P_n\}$, modeled by n timed automata $\mathscr{A} = \{\mathscr{A}_1, \ldots, \mathscr{A}_n\}$, respectively, where $\mathscr{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$, $1 \leqslant i \leqslant n$, and
- a set of m processors, $\mathscr{Q} = \{Q_1, \ldots, Q_m\}$.

Definition 6 (*Timed computation tree logic formula*). A timed computation tree logic formula has the following syntax.

$$\phi ::= \eta \mid \exists\Box\phi' \mid \exists\phi' \mathscr{U}_{\sim c} \phi'' \mid \neg\phi' \mid \phi' \vee \phi''. \tag{5}$$

Here, η is a mode predicate in $B(\cup_{i=1}^n C_i, \cup_{i=1}^n D_i)$, ϕ', ϕ'' are TCTL formulae, $\sim \in \{<, \leqslant, =, \geqslant, >\}$, and $c \in \mathscr{N}$. $\exists\Box\phi'$ means there exists a computation, from the current state, along which ϕ' is

always true. $\exists\phi'\mathcal{U}_{\sim c}\phi''$ means there exists a computation, from the current state, along which ϕ' is true until ϕ'' becomes true, within the time constraint of $\sim c$. Traditional shorthands like $\exists\Diamond$, $\forall\Diamond$, $\forall\Diamond$, $\forall\mathcal{U}$, $\wedge$, and $\rightarrow$ can all be defined [15]. □

Besides a system model and a property specification, since we are verifying *embedded* software, we also need the schedules generated by an embedded software synthesis procedure (see Section 2). Recall from Section 4.2, a set of schedules is denoted by: $Z = \{\zeta_k \mid \zeta_k = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_r}\rangle, P_{k_j} \in \mathscr{P}, 1 \leqslant j \leqslant r \leqslant n, 1 \leqslant k \leqslant n\}$, where $\zeta_k = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_r}\rangle$ is a schedule for processor Q_k, such that processes $P_{k_1}, P_{k_2}, \ldots, P_{k_r}$ are scheduled to run on Q_k, $r > 0$.

We are now ready to formulate our problem.

Definition 7 (*CERTS verification problem*). Given a real-time system $\mathscr{S} = \langle\mathscr{P}, \mathscr{Q}\rangle$, a TCTL formula ϕ, and a set of schedules Z, CERTS verification problem is to verify if $\mathscr{S}$ satisfies ϕ under the schedule Z. In notations, this is represented as $\mathscr{S} \models_Z \phi$. □

A model checking solution to the CERTS verification problem is proposed in this Section. Two model checking algorithms are given in Tables 2 and 3. The former is based on the processor-oriented verification approach presented in Section 4.3, while the latter is based on the process-oriented verification approach presented in Section 4.4. The main difference in these two algorithms is in the set of TA, which is input to the **Symbolic_MCheck()** procedure (Table 4).

In the *processor-oriented* verification approach, as given in Table 2, a set of TA, $\mathscr{B}$, is constructed, from the system description and from the set of schedules (generated from a synthesis method), to model the set of processors and this set is input to the symbolic model checking procedure. The construction procedure (**Construct_Processor_TA**()) was described in Section 4.3.

In the *process-oriented* verification approach, as given in Table 3, a set of TA, $\mathscr{A}'$, is constructed by modifying the set of process TA, $\mathscr{A}$, given in the system description and this set is input to the symbolic model checking procedure. The construction procedure (**Modify_Process_TA**()) was described in Section 4.4.

The symbolic model checking procedure (**Symbolic_MCheck**()) used in the two algorithms (Tables 2 and 3) is given in Table 4. A *region* is defined symbolically as a collection of states that satisfy a symbolic condition on clock variable values and a symbolic condition on discrete variable values. Given a region R, its symbolic clock condition and symbolic discrete variable condition are represented by *R.ClockCond* and *R.DVarCond*, respectively. In most model checking tools, *Difference Bound Matrices* (DBM) [3,13] and *Binary*

Table 2
Model checking algorithm for embedded software (processor-oriented)

```
Model_Check_Embedded_Software1(𝒮, φ, Z)
system 𝒮 = ⟨𝒫, 𝒬⟩;   // P_i ∈ 𝒫 modeled by
//𝒜_i = (M_i, m_i^0, C_i, D_i, χ_i, E_i, τ_i, ρ_i) ∈ 𝒜
tctl formula φ;
schedule set Z;
{
  Let ℬ be an empty set of TA;
  For k = 1, ..., |𝒬| {
    ℬ_k = Construct_Processor_TA(𝒜, ζ_k);
        // where ℬ_k is a timed automaton.
    ℬ = ℬ ∪ {ℬ_k};
  }
  Symbolic_MCheck(ℬ, φ);
}
```

Table 3
Model checking algorithm for embedded software (process-oriented)

```
Model_Check_Embedded_Software2(𝒮, φ, Z)
system 𝒮 = ⟨𝒫, 𝒬⟩;   // P_i ∈ 𝒫 modeled by
//𝒜_i = (M_i, m_i^0, C_i, D_i, χ_i, E_i, τ_i, ρ_i) ∈ 𝒜
tctl formula φ;
schedule set Z;
{
  Let 𝒜' be an empty set of TA;
  For k = 1, ..., |𝒫| {
    𝒜'_k = Modify_Process_TA(𝒜_k, Z);
        // where 𝒜'_k is a timed automaton.
    𝒜' = 𝒜' ∪ {𝒜'_k};
  }
  Symbolic_MCheck(𝒜', φ);
}
```

Table 4
Symbolic model checking procedure

```
Symbolic_MCheck(𝒜, φ)
set of TA 𝒜;     // 𝒜_i = (M'_i, m'^0_i, C'_i, D'_i, γ'_i, E'_i, τ'_i, ρ'_i) ∈ 𝒜, i ⩾ 1
tctl formula φ;
{
   Let Reach = Unvisited = {R_init};
        // Reach, Unvisited: Set of Regions,
        // R_init : r'^0_1 × r'^0_2 × ... × r'^0_|𝒜|,
// where r'^0_i is an initial region of 𝒜_i.
   While (Unvisited ≠ NULL) {
     R' = Dequeue(Unvisited);    // R': a region
     For all out-going transition, e, of R' {
       R'' = Successor_Region(R', e);    // R'': a region
       If R'' is consistent and R'' ∉ Reach {
         Reach = Reach ∪ {R''};
         Queue(R'', Unvisited);
         }
     }
   }
   Label_Region(Reach, φ);
   Return L(R_init);    // where L is the label assigned by
// Label_Region();
}
```

Table 5
Successor region function

```
Successor_Region(R, e)
region R;
transition e;
{
     R' = New_Region();
     R'.ClockCond = Advance(R.ClockCond)∧
     e.Trigger ∧ R.ClockCond;
     R'.ClockCond = Assign(R'.ClockCond, e.Assign);
     R'.ClockCond = R'.ClockCond∧
     (⋀_i R'. SubRegion_i.ClockCond(R'));
     R'.DVarCond = Assign(R.DVarCond, e.Assign);
     Return R';
}
```

Decision Diagrams (BDD) [12] are used to implement the symbolic clock and discrete variable conditions, respectively. Details on DBM and BDD can be found in the references. Discussion on them are out of scope here.

In the symbolic model checking procedure, two data-structures are maintained: a queue of regions (*Unvisited*) and a set of reachable regions (*Reach*). The former keeps a record of which regions are yet to be explored, while the latter keeps a record of all the regions reached. The procedure starts from an initial region, R_{init}, which is a cartesian product of the initial modes of all the TA in the input set of TA, $\mathscr{B}$. Initially, the initial region is queued in *Unvisited* and recorded in *Reach*. A region, R', is dequeued from *Unvisited* and corresponding to each out-going transition, e, of R' a successor region, R'', is constructed by the function **Successor_Region**(R', e) (see Table 5). If R'' is consistent and is not already in *Reach*, then it is recorded in *Reach* and queued in *Unvisited* for further exploration of its successors. The procedure loops until all regions in the queue have been explored. Finally, the regions in *Reach* are labeled according to the labeling algorithm **Label_Reach**$(Reach, \phi)$ (see Table 6), where ϕ is a TCTL formula, such that $\mathscr{S} \models_Z \phi$ is to be verified. This procedure finally outputs the label that has been assigned to the initial region, R_{init}.

As detailed in Table 5 (**Successor_Region**()), the successor region is constructed as follows. Given a region R and an out-going transition e, the successor region R' is constructed by first advancing (**Advance**()) all clock values till it satisfies the triggering condition ($e.Trigger$) of e, while at the same time still satisfying the clock condition of R, $R.ClockCond$. This first step gives an intermediate symbolic clock condition $R'.ClockCond$ for the successor region R'. Second, the clock resets in $e.Assign$ are applied to $R'.ClockCond$ by **Assign**(). Third, the clock conditions of all sub-regions of R' have also to be satisfied by $R'.ClockCond$. Finally, the discrete variable values are assigned to $R.DVarCond$ to obtain the new symbolic condition $R'.DVarCond$. In this way, both the clock and discrete variable symbolic conditions of the successor region R' are thus computed.

The labeling algorithm, **Label_Region()**, is presented in Table 6. This algorithm assigns a label, $L(R, \phi)$, to each region, R, in the set of regions *RSet*. The label indicates if the region R satisfies ϕ. This labeling is computed as follows. For a mode predicate (see Definition 1), the label is *true* if the region satisfies the mode predicate and it is *false* otherwise. For a TCTL path formula, ϕ, the label is computed recursively according to the semantics of the formula.

Table 6
Label region function

Label_Region($RSet, \phi$)
set of regions RSet;
tctl formula ϕ;
{
For each $R \in RSet$, calculate recursively the label of R, $L(R)$, as follows:
case $\phi = x \sim c$:
$L(R,\phi) :=$ true; if $x \sim c$ is true in R,
false; otherwise
case $\phi = x - y \sim c$:
$L(R,\phi) :=$ true; if $x - y \sim c$ is true in R,
false; otherwise
case $\phi = d \sim c$:
$L(R,\phi) :=$ true; if $d \sim c$ is true in R,
false; otherwise
case $\phi = \eta_1 \wedge \eta_2$:
$L(R,\phi) :=$ true; if both η_1 and η_2 are true in R,
false; otherwise
case $\phi = \neg\eta_1$:
$L(R,\phi) :=$ true; if η_1 is false in R,
false; otherwise
case $\phi = \exists\Diamond\phi' \mathscr{U}_{\sim c}\phi''$:
$L(R,\phi) :=$ true; if there is a successor R' of R such that $L(R',\phi'')$ is true, and
there is a path, π, from R to R' such that for all regions R'' along π,
$L(R'',\phi')$ is true and time$(R,R') \sim c$ is true, where time (R,R')
is the amount of time for going from a state in R to a state in R',
along path π.
false; otherwise
Similarly for the other cases: $\phi = \exists\Box\phi' \mathscr{U}_{\sim c}\phi''$, $\phi = \forall\Diamond\phi' \mathscr{U}_{\sim c}\phi''$, and $\phi = \forall\Box\phi' \mathscr{U}_{\sim c}\phi''$.
}

The symbolic model checking procedure presented above verifies if the input set $\mathscr{B}$ of TA satisfies the given TCTL formula ϕ. Application examples are given in Section 6 to illustrate and compare the two proposed verification approaches and also how they compare with traditional approaches.

6. Examples

Several application examples are given here to illustrate our proposed model checking procedure for the verification of concurrent embedded real-time software. First, we will illustrate the SVM approach proposed in Section 3.2 and how it compares with the conventional VSM approach described in Section 3.1. Second, we will illustrate the two verification approaches presented in Sections 4.3 and 4.4. Comparisons are also made between the two proposed approaches. It illustrates the advantage of verifying under system concurrency, compared to verifying under process concurrency.

In the following experiments, we use the *State-Graph Manipulators* (SGM) [17,18], a high-level real-time system verification tool based on the model checking framework. SGM allows flexibility in comparing different verification techniques, such as reduction technique, scheduling technique, etc. Hence, we used SGM. All the experiments were run on a Sun 296 MHz UltraSPARC-II workstation with 256 MB physical memory.

6.1. SVM approach versus VSM approach

The proposed SVM approach was compared analytically with the conventional VSM approach

in Section 3.3. Here, we will illustrate through application examples the actual comparison between the two verification approaches.

6.1.1. Fischer's mutual exclusion protocol

The first example is the *Fischer's Mutual Exclusion Protocol* (FMEP) [1,19,20] as first introduced in Section 4.1 and illustrated by a process timed automaton in Fig. 2. In this example, each process tries to enter a critical section (M_4) by checking to determine whether the value of variable lock is zero, writing its index (i) to lock within less than one time unit after lock is read as zero, and then re-checking if the value of lock is still its index after one time unit. The read time should be less than the write time for the protocol to work properly. This is exactly the property to be verified. In Fig. 2, x_i is a clock variable and *lock* is a discrete variable.

The size of state-space of a system of processes obeying the FMEP increases exponentially due to a drastic increase in the number of possible concurrencies. This is observable from the *non-scheduled* rows in Table 7. The number of modes and transitions represent the state-space sizes that the conventional VSM approach needs to explore for verification. Here, n, the number of processes in the system, was varied from 2 to 6 to observe the effect of increasing concurrency on the state-space sizes. For illustrating SVM, a system of process timed automata was scheduled by assigning priorities to concurrent lock variable writes (transition $M_2 \rightarrow M_3$). The results of scheduling FMEP processes are given by the *scheduled* rows in Table 7. On comparison, we observe a large difference between the state-space sizes explored by the two approaches. Thus, verification scalability is improved when the SVM approach is adopted.

6.1.2. Signal polling system

The second example is the *signal polling system* (SPS) as first introduced in Section 4.2 and illustrated by a process timed automaton in Fig. 3. This example illustrates not only how SVM explores a smaller state-space compared to VSM, but also how different scheduling techniques affect the sizes of state-spaces explored for verification.

This application is a distributed signal polling system that is generally located at each entry/exit gates of a parking lot. In this example, each process initializes a counter to 500 vacant parking spaces. Then, it starts to poll for any Car-Entry, Car-Exit, or Check-Count signal. When a signal is detected, appropriate actions are carried out. The counter value is decremented for a Car-Entry signal and incremented for a Car-Exit signal. The counter value is output for a Check-Count signal. After completing actions, the polling process is repeated. Here, we need to verify that a car is never allowed entry when there are no vacant parking space available ($Count = 0$).

Experiments were carried out for this example with and without scheduling, the details of which are shown in Table 8. Verification of a 4-process signal polling system required exploring 78K modes with 205K transitions when no scheduling was applied. This is a very large state-space, the construction of which requires large amounts of

Table 7
SVM versus VSM: Fischer's mutual exclusion protocol

n	Scheduled	Approach	#Modes	#Trans	Memory (MB)	Time (s)
2	No	VSM	23	38	0.78	0.08
2	Yes	SVM	**8**	**9**	0.78	0.05
3	No	VSM	103	249	0.97	0.57
3	Yes	SVM	**22**	**28**	0.88	0.27
4	No	VSM	467	1532	2.28	7.31
4	Yes	SVM	**82**	**115**	1.81	3.65
5	No	VSM	2381	10,065	10.47	113.00
5	Yes	SVM	**392**	**767**	10.40	68.25
6	No	VSM	14,181	74,046	98.02	2487.00
6	Yes	SVM	**2284**	**3589**	88.48	1610.22

memory (178 MB) and time (9608 s). Three scheduling techniques were applied to this example: *post-signal* scheduling, *pre-signal* scheduling, and both *post-signal* and *pre-signal* scheduling. Post-signal scheduling is scheduling of the processes that have detected signals concurrently. Pre-signal scheduling is scheduling of processes before any signal detection is started. We observe from Table 8 that the three types of scheduling techniques result in different sizes of state-spaces. Applying both post- and pre-signal scheduling results in the smallest state-space. Applying pre-signal scheduling results in a smaller state-space than applying post-signal scheduling. This is consistent with our intuition, because pre-signal scheduling applies a much greater restriction on the behavior of the processes than post-signal scheduling.

6.2. System concurrency versus process concurrency

Instead of process concurrency (number of processes), verification under system concurrency (number of processors executing software) was proposed in Section 4. Two different models were also proposed, namely, processor-oriented and process-oriented in Sections 4.3 and 4.4, respectively. Here, verification under system concurrency is performed for two application examples and both the models compared with the conventional verification under process concurrency.

6.2.1. Fischer's mutual exclusion protocol

This example is a 4-process system obeying the *Fischer's Mutual Exclusion Protocol* (FMEP). See Section 6.1.1 for a short description on FMEP. Three different system configurations are considered: one processor, two processors, and four processors. The 4-process software system was executed on all the three system configurations and the state-space sizes recorded as shown in Table 9, where column n represents the number of processes and column m represents the number of processors.

It can be observed that compared to verifying under process concurrency (row 1 of Table 9), verifying under system concurrency (rows 2–5 of Table 9) results in a much smaller state-space and in a higher verification scalability. It can also be observed that a process-oriented model for verification under system concurrency has a smaller state-space compared to that of a processor-oriented model (rows 3 and 4 of Table 9).

6.2.2. Signal polling system

This example is a 4-process SPS, which was introduced in Section 6.1.2. Three different system configurations are considered: one processor, two processors, and four processors. The 4-process software system was executed on all the three system configurations and the state-space sizes recorded as shown in Table 10.

Table 8
SVM versus VSM: signal polling system

n	Schedule	Approach	#Modes	#Trans	Memory (MB)	Time (s)
4	No	VSM	78,347	205,578	178.12	9608.36
4	Post-signal	SVM	1018	1255	6.57	34.20
4	Pre-signal	SVM	29	41	10.69	82.19
4	Post/pre-signal	SVM	22	26	5.48	27.06

Table 9
System versus process concurrency: Fischer's mutual exclusion protocol

n	m	Schedule	Approach	Model	#Modes	#Trans	Memory (MB)	Time (s)
4	4	No	VSM	N/A	467	1532	2.28	7.31
4	4	Yes	SVM	Process	82	115	1.81	3.65
4	2	Yes	SVM	Processor	92	152	0.86	0.16
4	2	Yes	SVM	Process	8	8	0.79	0.16
4	1	Yes	SVM	Process	4	4	0.79	0.16

Table 10
System versus process concurrency: signal polling system

n	*m*	Schedule	Approach	Model	#Modes	#Trans	Memory (MB)	Time (s)
4	4	No	VSM	N/A	78,347	205,578	178.12	9608.36
4	4	Yes	SVM	Process	1018	1255	6.57	34.20
4	2	Yes	SVM	Processor	1327	2727	2.92	7.36
4	2	Yes	SVM	Process	57	79	1.05	0.51
4	1	Yes	SVM	Process	29	41	0.98	0.30

Similar to the previous example, it can be observed that compared to verifying under process concurrency (row 1 of Table 10), verifying under system concurrency (rows 2–5 of Table 10) results in a much smaller state-space and in a higher verification scalability. It can also be observed that a process-oriented model for verification under system concurrency has a smaller state-space compared to that of a processor-oriented model (rows 3 and 4 of Table 10).

7. Conclusion

A verification method for CERTS was proposed. The method covered three important verification issues: *when*, *where*, and *how* should CERTS verification be performed. In answer to the *when* issue, a SVM strategy was proposed. It was both analytically and experimentally validated how SVM supercedes conventional VSM and SMV approaches. In answer to the *where* issue, instead of the conventional verification under *process concurrency*, verification under *system concurrency* was proposed. The advantage of verifying under system concurrency was also illustrated through application examples. In answer to the *how* issue, a complete symbolic model checking procedure was presented within two different verification approaches: processor-oriented and process-oriented. Application examples show how the proposed answers to each of the three issues aid in CERTS verification.

References

[1] M. Abadi, L. Lamport, An old-fashioned recipe for real time, in: REX Workshop, Real-Time Theory in Practice, Lecture Notes in Computer Science, vol. 600, June 1991, pp. 1–27.

[2] K. Altisen, G. Gobler, A. Pneuli, J. Sifakis, S. Tripakis, S. Yovine, A framework for scheduler synthesis, in: Real-Time System Symposium (RTSS'99), IEEE Computer Society Press, 1999.

[3] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, H. Wong-Toi, An implementation of three algorithms for timing verification based on automata emptiness, in: Proceedings of IEEE International Conference on Real-Time Systems Symposium (RTSS'92), 1992.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, Model checking for real-time systems, in: Proceedings of IEEE Logics in Computer Science, 1990.

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

[6] R. Alur, C. Courcoubetis, T. Henzinger, P.-H. Ho, Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems, in: R. Grossman, A. Nerode, A. Raun, H. Rischel (Eds.), Hybrid Systems, vol. 736, Lecture Notes in Computer Science, Springer, Berlin, 1993, pp. 209–229.

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

[8] E. Asarin, O. Maler, A. Pneuli. Symbolic controller synthesis for discrete and timed systems, in: P. Antsaklis, W. Kohn, A. Nerode, S. Sastry (Eds.), Hybrid Systems II, vol. 999, Lecture Notes in Computer Science, Springer, Berlin, 1995, pp. 1–20.

[9] E. Asarin, O. Maler, A. Pneuli, J. Sifakis, Controller synthesis for timed automata, in: Proceedings of the System Structure and Control, IFAC, Elsevier, July 1998.

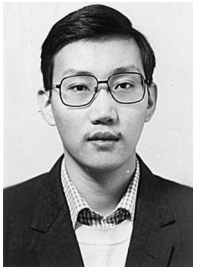
[10] F. Balarin, M. Chiodo, Software synthesis for complex reactive embedded systems, in: Proceedings of International Conference on Computer Design (ICCD'99), IEEE CS Press, October 1999, pp. 634–639.

[11] F. Balarin, et al., Hardware–software Co-design of Embedded Systems: the POLIS approach, Kluwer Academic Publishers, Dordrecht, 1997.

[12] R. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers C-35 (8) (1986).

[13] D. Dill, Timing assumptions and verification of finite-state concurrent systems, in: Proceedings of the International

Conference on Computer-Aided Verification, LNCS, vol. 407, Springer, Berlin, 1989.
[14] E. Emerson, Temporal and modal logic, Handbook of Theoretical Computer Science, 1990.
[15] T. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, in: Proceedings of the IEEE Logics in Computer Science, 1992.
[16] P.-A. Hsiung, Timing coverification of concurrent embedded real-time systems, in: Proceedings of the Seventh IEEE/ACM International Workshop on Hardware Software Codesign (CODES'99), ACM Press, New York, May 1999, pp. 110–114.
[17] P.-A. Hsiung, F. Wang, A state-graph manipulator tool for real-time system specification and verification, in: Proceedings of the Fourth International Conference on Real-Time Computing Systems and Applications (RTCSA'98), IEEE Computer Society Press, October 1998, pp. 181–188.
[18] P.-A. Hsiung, F. Wang, User-friendly verification, in: International Conference on Formal Description Techniques For Distributed Systems and Communication Protocols & Protocol Specification, Testing, and Verification (FORTE/PSTV'99), October 1999.
[19] L. Lamport, A fast mutual exclusion algorithm, ACM Transactions on Computer Systems 5 (1) (1987) 1–11.
[20] K.G. Larsen, B. Steffen, C. Weise, Fischer's protocol revisited: a simple proof using modal constraints, in: Hybrid System III, Lecture Notes in Computer Science, vol. 1066, 1996, pp. 604–615.
[21] B. Lin, Efficient compilation of process-based concurrent programs without run-time scheduling, in: Proceedings of Design Automation and Test Europe (DATE'98), ACM Press, New York, February 1998, pp. 211–217.
[22] B. Lin, Software synthesis of process-based concurrent programs, in: Proceedings of Design Automation Conference (DAC'98), ACM Press, New York, June 1998, pp. 502–505.
[23] O. Maler, A. Pnueli, J. Sifakis. On the synthesis of discrete controllers for timed systems, in: E. Mayr, C. Puech (Eds.), Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95), vol. 900, Lecture Notes in Computer Science, Springer, Berlin, March 1995, pp. 229–242.
[24] P. Merlin, G. Bochman, On the construction of submodule specifications and communication protocols, ACM Transactions on Programming Languages and Systems 5 (1) (1983) 1–25.
[25] M. Sgroi, L. Lavagno, Y. Watanabe, A. Sangiovanni-Vincentelli, Synthesis of embedded software using free-choice petri nets, in: Proceedings of the Design Automation Conference (DAC'99). ACM Press, New York, June 1999.
[26] H. Wong-Toi, G. Hoffman, The control of dense real-time discrete event systems. Technical report STAN-CS-92-1411, Stanford University, 1992.
[27] X. Zhu, B. Lin, Compositional software synthesis of communicating processes. in: Proceedings of International Conference on Computer Design (ICCD'99), IEEE CS Press, October 1999, pp. 646–651.



Dr. Pao-Ann Hsiung received the B.S. degree in mathematics and the Ph.D. degree in electrical engineering from the National Taiwan University (NTU), 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, NTU. From 1996 to 2001, he was a post-doctoral research associate at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC. Starting from 2001, he joined the faculty of the Department of Computer Science and Information Engineering, National Chung-Cheng University, Chiayi, Taiwan, ROC. Dr. Hsiung is a member of the IEEE and the IEEE Computer Society. He has been included in several professional listings such as Marquis' Who's Who in the World (17th Millenium Edition, 2000), Outstanding People of the 20th Century (2nd Edition, 2000, Cambridge, England), Who's Who in Formal Methods, ACM SIGDA's design automation professionals,... Dr. Hsiung is on the program committee of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), International Workshop on Real-Time Constraints (RTC'99), International Workshop on Distributed System Validation and Verification (DSVV'2000), and 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000). He served as session organizer and chair for PDPTA'99, as workshop organizer and chair for RTC'99, and as workshop organizer and chair for DSVV'2000. He has published more than 40 papers in international journals and conferences. He has been taking an active part in paper refereeing for international journals and conferences. His main research interests include: hardware–software codesign and coverification, real-time system specification and verification, system-level design automation of multiprocessor systems, parallel architecture design and simulation, and object-oriented design techniques in system syntheses.