Concurrent Embedded Real-Time Software Verification

Pao-Ann Hsiung
Institute of Information Science, Academia Sinica, Taiwan, ROC
E-mail: hpa@computer.org

Abstract

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 Concurrent Embedded Real-Time Software *(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. Application examples illustrate the usefulness of our technique in increasing verification scalability.*

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. 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) [17], which 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. Verification can be performed at three different points: before scheduling, after scheduling, and after code generation. We propose to verify software *after scheduling* and *before code generation*, as discussed in Section 3.

Where should software be verified? There are two kinds of concurrencies in a hardware-software system: *system concurrency* and *process concurrency*. System concurrency is the number of CPUs running software. Process concurrency is the number of concurrent processes. Conventionally, software is verified under process concurrency. We propose that embedded software should instead be verified under system concurrency, which significantly increases verification efficiency and scalability, as 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) [5] and a property is specified in *Timed Computation Tree Logic* (TCTL) [9, 11]. We propose two model checking algorithms for *Concurrent Embedded Real-Time Software* (CERTS). 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. This is described in Section 5.

Section 2 gives a brief survey of current software synthesis methods. 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 *system concurrency*. Section 5 answers the *how* question by proposing two verification algorithms for CERTS. Section 6 gives two application examples. Section 7 concludes with some future research directions.

2. Embedded Software Synthesis

Currently, *software synthesis* is a hot topic of research in the field of hardware-software codesign. Partial software synthesis was performed for communication protocols [16], plant controllers [6], and real-time schedulers [2]. Only recently has there been some work on automatically generating software code for embedded systems [15, 17, 18]. As far as the authors know, no automatic software synthesis method is available for *concurrent real-time* embedded software. In the following, we will survey the existing works on the synthesis of non real-time software.

Lin [15] proposed an algorithm that generates a software program from a concurrent process specification through in-

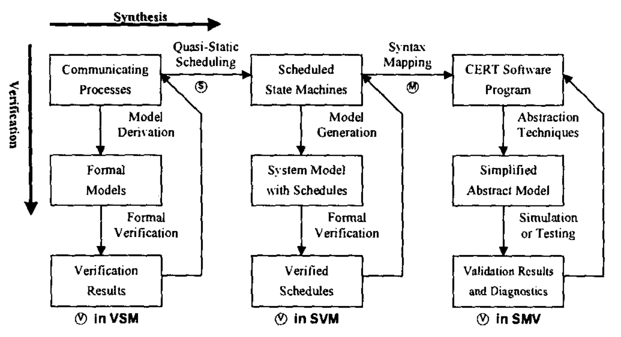

Figure 1. En-route Verification

termediate 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 the model is always schedulable. The proposed method applies *quasi-static scheduling* 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. Later, Zhu and Lin [18] 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. [17]. A quasi-static scheduling algorithm was proposed for *Free-Choice Petri Nets* (FCPN) [17]. 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. All the above work suggest that 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, just as we did for hardware-software codesign [12, 10].

3. Schedule-Verify-Map Strategy

This section answers the *when* question, that is, *"when should software be verified?"* As depicted in Figure 1, there are three stages in synthesis (first row in the figure): *process specification*, *scheduling*, and *code generation*.

- **Stage (1).** In process specification, a set of communicating processes representing the behavior of desired software is specified, which can be in the form of a set of Petri Nets [15, 18, 17], or in formal specification languages such as Esterel, LOTOS, etc.
- **Stage (2).** In scheduling, except for run-time dependent computations, all other computations in the specified processes are quasi-statically scheduled [15, 17]. 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 software code. A software time loop is utilized to maintain the schedule in the finite state-machines.

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 Figure 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 Figure 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 by 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.

Verification of 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.

3.2. Proposed SVM Approach

To overcome the difficulties in verification presented in the previous subsection, we propose a new approach called *Schedule-Verify-Map* (SVM). In SVM, verification is performed after scheduling and before code generation. Since scheduling eliminates certain regions in the state-space, SVM will explore a much smaller part of the state-space. 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 three approaches — *Verify-Schedule-Map* (VSM) adopted by verification scientists, *Schedule-Map-Verify* (SMV) adopted by design engineers, and our proposed *Schedule-Verify-Map* (SVM) approach, we have the pros and cons of each summarized in Table 1.

Table 1. Verification Approach Comparison

	Correct	Feasible	State-Space	Complete
VSM	Too Sure	Vaguely	Exp. Large	Over
SVM	Sure	Largely	Reduced	Yes
SMV	Not Sure	Practically	Medium	No

In the following, the sets of integers and non-negative real numbers are denoted by $\mathcal{N}$ and $\mathcal{R}_{\geq 0}$, respectively.

A timed automaton (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 $x, y \in C$, $\sim \in \{\leq, <, =, \geq, >\}$, $c \in \mathcal{N}$, $d \in D$, and η_1, η_2 are mode predicates. ||

Let $B(C, D)$ be the set of all mode predicates over C and D.

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 discrete variables to specific integer values. ||

4. Handling Concurrency

This section answers the *where* question, that is, "*where should software be verified*?" In the rest of this section, we propose to verify embedded software under system concurrency, rather than under process concurrency.

4.1. Verification under Two 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, a three-process system has 1239 modes and 4013 transitions, and a four-process system has approximately 28K modes and 120K transitions. The increase is drastic.

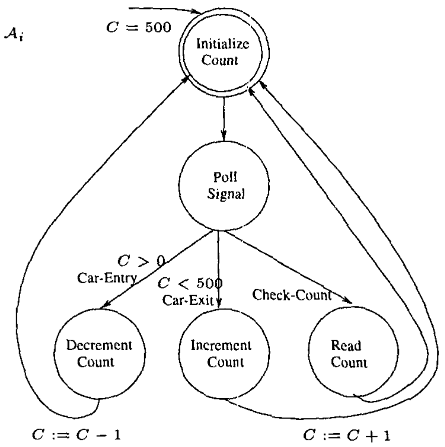

Figure 2. Signal Polling System

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 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 four-process system verified under process concurrency of four (78K modes and 205K transitions). A timed automaton for the signal polling system is given in Figure 2.

Unless mentioned otherwise, assume we are given a system $\mathcal{S}$ with n processes $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, modeled by n timed automata $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$, respectively, where $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i)$, $1 \leq i \leq n$. Also, assume there are m processors in system $\mathcal{S}$, that is, $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_m\}$. Hence, a system is defined as a two-tuple $\mathcal{S} = \langle \mathcal{P}, \mathcal{Q} \rangle$.

On software synthesis, the n processes in $\mathcal{P}$ are quasi-statically scheduled on the m processors in $\mathcal{Q}$ (refer to Section 2 for quasi-static scheduling). Let Z be the set of valid schedules generated by any software synthesis method, that is, $Z = \{\zeta_i \mid \zeta_i = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_r} \rangle, P_{k_j} \in \mathcal{P}, 1 \leq j \leq r \leq n, 1 \leq i \leq m\}$, 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 $P_{k_1}, \ldots, P_{k_r}$

are scheduled to run on Q_i.

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

4.2. Processor-Oriented Verification

An intuitive method is to create a new timed automaton for modeling the behavior of each processor. This is called *processor-oriented verification.* Besides being intuitive, it can be easily extended to include process preemptions.

Based on the syntax representation of a timed automaton, we know that each process automaton, $\mathcal{A}_i$, either has a transition, $e_f \in E_i$, that loops back to the initial mode, m_i^0, from some mode in $M_i \backslash \{m_i^0\}$ 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 out-going transition.

A processor timed automaton, $\mathcal{B}_i = (M_i', m_i'^0, C_i', D_i', \chi_i', E_i', \tau_i', \rho_i')$, 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 $\mathcal{A}_{k_j}$ into $\mathcal{B}_i$. The inclusion method involves how two consecutive TA, $\mathcal{A}_{k_j}$ and $\mathcal{A}_{k_{j+1}}$ are to be merged into the new $\mathcal{B}_i$. 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 $\mathcal{A}_{k_{j+1}}$. 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 $\mathcal{A}_{k_{j+1}}$. Thus, transitions e_f and e_f' interconnect the two consecutive TA, $\mathcal{A}_{k_j}$ and $\mathcal{A}_{k_{j+1}}$ in the new TA $\mathcal{B}_i$. Suppose a partial schedule, $\overset{u}{\overline{P_{k_j}, \ldots, P_{k_{j+v}}}}$, is to be looped for u times, where $u > 1$, $v \geq 0$, 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 $\mathcal{A}_{k_{j+v}}$ with the initial mode of $\mathcal{A}_{k_v}$ and with the initial mode of the next process after the loop in a schedule.

4.3. 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 $\mathcal{A}$. This approach is called *process-oriented verification.* This approach does not allow process preemption.

A *processor locking variable* is used to restrict the execution of a process according to a schedule. It 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 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, $1 \leq k \leq m$. Suppose the processor schedules are as follows: $\zeta_k = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_r} \rangle$, $1 \leq k \leq m$, $1 \leq r \leq n$. Let the initial value of l_k be k_1. Assume that process P_{k_j} is scheduled on processor Q_k, $1 \leq j \leq r$. Modify each process timed automaton, $\mathcal{A}_{k_j}$, as follows:

- Create a new initial mode, $m_{k_j}'^0$, for $\mathcal{A}_{k_j}$,
- Create a new transition, e^0, from $m_{k_j}'^0$ to $m_{k_j}^0$,
- Let the triggering condition $\tau_{k_j}(e^0)$ be $l_k = k_j$,
- For each looping transition, e, let $\rho_{k_j}(e) = \rho_{k_j}(e); (l_k := k_{j+1})$,
- For each loop repetition in a schedule, a counter variable maintains the loop execution count.

5. Model Checking CERTS

The framework of verification that we use for software verification is the popular *model checking* framework [4, 11], 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 *Timed Automata* (TA) [5] (see Definition 2) and a temporal property is specified using *Timed Computation Tree Logic* (TCTL) [9, 11].

Definition 3 : <u>TCTL Formula</u>
A timed computation tree logic formula has the following syntax: $\phi ::= \eta \mid \exists\Box\phi' \mid \exists\phi'\mathcal{U}_{\sim c}\phi'' \mid \neg\phi' \mid \phi' \vee \phi''$. Here, η is a mode predicate in $\mathcal{B}(\cup_{i=1}^n C_i, \cup_{i=1}^n D_i)$, ϕ', ϕ'' are TCTL formulae, $\sim \in \{<, \leq, =, \geq, >\}$, and $c \in \mathcal{N}$. $\exists\Box\phi'$ 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 $\exists\Diamond$, $\forall\Box$, $\forall\Diamond$, $\forall\mathcal{U}$, $\wedge$, and $\rightarrow$ can all be defined [11]. ||

We will now formulate our problem.

Definition 4 : <u>CERTS Verification Problem</u>
Given a real-time system $\mathcal{S} = \langle \mathcal{P}, \mathcal{Q} \rangle$, a TCTL formula ϕ, and a set of schedules Z, *Concurrent Embedded Real-Time Software (CERTS) verification problem* is to verify if $\mathcal{S}$ satisfies ϕ under the schedule Z. In notations, this is represented as $\mathcal{S} \models_Z \phi$. ||

Two model-checking algorithms are proposed to solve the CERTS verification problem: *processor-oriented* and *process-oriented.* In the *processor-oriented* verification approach, as given in Table 2, a set of TA, $\mathcal{B}$, 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

Table 2. Model Checking Algorithm for Embedded Software (Processor-Oriented)

```
**Model_Check_Embedded_Software1**$(\mathcal{S}, \phi, Z)$
*system* $\mathcal{S} = \langle \mathcal{P}, \mathcal{Q} \rangle$;    // $P_i \in \mathcal{P}$ modeled by
// $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, \chi_i, E_i, \tau_i, \rho_i) \in \mathcal{A}$
*tctl formula* $\phi$;
*schedule set* $Z$;
{ Let $\mathcal{B}$ be an empty set of TA;
  **For** $k = 1, \ldots, |\mathcal{Q}|$ {
    $\mathcal{B}_k$ = **Construct_Processor_TA**$(\mathcal{A}, \zeta_k)$;
          // where $\mathcal{B}_k$ is a timed automaton.
    $\mathcal{B} = \mathcal{B} \cup \{\mathcal{B}_k\}$; }
  **Symbolic_MCheck**$(\mathcal{B}, \phi)$; }
```

Table 3. Symbolic Model Checking Procedure

```
**Symbolic_MCheck**$(\mathcal{B}, \phi)$
*set of TA* $\mathcal{B}$;
// $\mathcal{B}_i = (M_i', m_i'^0, C_i', D_i', \chi_i', E_i', \tau_i', \rho_i') \in \mathcal{B}, i \geq 1$
*tctl formula* $\phi$;
{ Let $Reach = Unvisited = \{R_{init}\}$;
  **While** $(Unvisited \neq NULL)$ {
    $R'$ = **Dequeue**$(Unvisited)$;    // $R'$: a region
    **For** all out-going transition, $e$, of $R'$ {
      $R''$ = **Successor_Region**$(R', e)$;
      **If** $R''$ is consistent and $R'' \notin Reach$ {
        $Reach = Reach \cup \{R''\}$;
        **Queue**$(R'', Unvisited)$; } } }
  **Label_Region**$(Reach, \phi)$;
  **Return** $L(R_{init})$; }
```

checking procedure. The construction procedure (**Construct_Processor_TA**()) was described in Section 4.2. The *process-oriented* verification approach is very similar to the processor-oriented approach, except it requires a modification of the process TA, as described in Section 4.3.

The symbolic model checking procedure (**Symbolic_MCheck**()) used in the algorithm (Table 2) is given in Table 3. 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* [3, 8] and *Binary Decision Diagrams* [7] are used to implement symbolic clock and discrete variable conditions, respectively. Due to page-limits, **Successor_Region**() and **Label_Reach**() procedures are left out.

Table 4. Fischer's Mutual Exclusion Protocol

n	Sch	App	#M	#T	Mem	Time
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	1,532	2.28	7.31
4	Yes	SVM	**82**	**115**	1.81	3.65
5	No	VSM	2,381	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	**2,284**	**3,589**	88.48	1610.22

Units: Memory (**Mem**) in MB and Time in seconds
n = #Processes, **Sch** = Scheduled, **App** = Approach,
#M = No. of Modes, **#T** = No. of Transitions

6. Examples

All the experiments were run on a Sun 296 MHz UltraSPARC-II workstation with 256 MB memory.

6.1. SVM Approach v/s VSM Approach

The proposed *Schedule-Verify-Map* (SVM) approach was compared analytically with the conventional *Verify-Schedule-Map* (VSM) approach in Section 3.1. We will illustrate through application example the actual comparison between the two verification approaches. This example is the *Fischer's Mutual Exclusion Protocol* (FMEP) [1, 13, 14].

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 4. Here, n, the number of processes, 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. The results of scheduling FMEP processes are given by the *scheduled* rows in Table 4. There is a large difference between the state-space sizes explored by the two approaches. Verification scalability is improved when the SVM approach is adopted.

6.2. System v/s Process Concurrency

Instead of process concurrency (number of processes), verification under system concurrency (number of processors executing software) was proposed in Section 4. Two

Table 5. Signal Polling System

m	Sch	App	L	#M	#T	Mem	Time
4	No	VSM	-	78K	205K	178	9608
4	Yes	SVM	P	1,018	1,255	7	34
2	Yes	SVM	Q	1,327	2,727	3	7
2	Yes	SVM	P	57	79	1	0.5
1	Yes	SVM	P	29	41	1	0.3

L = Orientation Model, P = Process, Q = Processor

different models were also proposed, namely, processor-oriented and process-oriented in Sections 4.2 and 4.3, respectively. Here, verification under system concurrency is performed for two application examples and both the models compared with the conventional verification under process concurrency.

This example is a 4-process signal polling system (SPS), which was introduced in Section 4.1. Three different system configurations are considered: 1 processor, 2 processors, and 4 processors. The 4-process software system was executed on all the three system configurations and the state-space sizes recorded as shown in Table 5.

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

7. Conclusion

A verification method for *Concurrent Embedded Real-Time Software* (CERTS) was proposed, answering three important verification issues: *when*, *where*, and *how* should CERTS be verified. In answer to the *when* issue, a *Schedule-Verify-Map* (SVM) strategy was proposed. 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 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. Examples show how the proposed answers to each of the three issues aid in CERTS verification.

References

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

[2] K. Altisen, G. Gobler, A. Pneuli, J. Sifakis, S. Tripakis, and 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, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proc. IEEE Intl. Conf. Real-Time Systems Symposium (RTSS'92)*, 1992.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, and D. Dill. Model checking for real-time systems. In *Proc. IEEE Logics in Computer Science*, 1990.

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

[6] E. Asarin, O. Maler, A. Pneuli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*. IFAC, Elsevier, July 1998.

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

[8] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. International Conference on Computer-Aided Verification, LNCS*, volume 407. Springer Verlag, 1989.

[9] E. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, 1990.

[10] J.-M. Fu, T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen. Hardware-software timing coverification of distributed embedded systems. *IEICE Trans. on Information and Systems*, to appear.

[11] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proc. IEEE Logics in Computer Science*, 1992.

[12] P.-A. Hsiung. Hardware-software timing coverification of concurrent embedded real-time systems. *IEE Proceedings on Computers and Digital Techniques*, 147(2):81–90, March 2000.

[13] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, February 1987.

[14] K. G. Larsen, B. Steffen, and C. Weise. Fischer's protocol revisited: A simple proof using modal constraints. In *Hybrid System III, Lecture Notes in Computer Science*, volume 1066, pages 604–615, 1996.

[15] B. Lin. Software synthesis of process-based concurrent programs. In *Proc. of Design Automation Conference (DAC'98)*, pages 502 – 505. ACM Press, June 1998.

[16] P. Merlin and G. Bochman. On the construction of submodule specifications and communication protocols. *ACM Trans. on Programming Languages and Systems*, 5(1):1 – 25, January 1983.

[17] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proc. Design Automation Conference (DAC'99)*. ACM Press, June 1999.

[18] X. Zhu and B. Lin. Compositional software synthesis of communicating processes. In *Proc. of International Conference on Computer Design (ICCD'99)*, pages 646 – 651. IEEE CS Press, October 1999.