PAPER Special Section on Smart Multimedia & Communication Systems

Model Checking of Real-Time Properties of Resource-Bound Process Algebra

Junkil PARK[†], *Student Member*, **Jungjae LEE**[†], **Jin-Young CHOI**[†a)], *and* **Insup LEE**[††], *Nonmembers*

SUMMARY The algebra of communicating shared resources (ACSR) is a timed process algebra which extends classical process algebras with the notion of a resource. In analyzing ACSR models, the existing techniques such as bisimulation checking and Hennessy-Milner Logic (HML) model checking are very important in theory of ACSR, but they are difficult to use for large complex system models in practice. In this paper, we suggest a framework to verify ACSR models against their requirements described in an expressive timed temporal logic. We demonstrate the usefulness of our approach with a real world case study.

key words: *ACSR, model checking, action-based modeling, real-time temporal logic, resource-bound process algebra*

1. Introduction

The Algebra of Communicating Shared Resources, ACSR [1], is a timed process algebra that is based on the premise that the timed behavior of a real-time system is affected not only by the time its components take to execute and synchronize, but also by delays introduced due to the scheduling of actions that compete for shared resources.

The ACSR computation model is based on the view that the components of a real-time system execute synchronously time-and-resource consuming actions and communicate through instantaneous events asynchronously, except when two components synchronize through matching events. To be able to specify real-time systems accurately, ACSR supports static priorities that can be used to arbitrate between actions competing for shared resources and between events that are ready for synchronization.

ACSR models can be analyzed by bisimulation checking and Hennessy-Milner Logic model checking [2]. The bisimulation checking approach for verifying ACSR models is showing that the models are bisimilar to their requirement specifications. For large and complex systems, it is hard to model requirement specifications using ACSR because it usually involves sequentializing the behaviors of parallel processes to inspect that the specification satisfies all requirements of the system, and it doesn't contain undesirable behaviors. On the other hand, in the existing model checking approach of ACSR, HML has not enough expressiveness power and it is too complicated to describe requirements of real-time systems in practice.

ACSR needs a temporal logic to expressively describe both untimed and real-time properties; and a method that can check ACSR models against the properties. We suggest the method that analyzes ACSR models using model checking technique by (1) translating (abstracting) a Timed LTS to Abstracted Timed LTS, (2) extending ACTL* to Bounded ACTL* by adding bounded temporal operators in order to easily describe timing properties, and (3) using a toolchain composed by VERSA and CWB-NC [3] that provide effective ACTL* model checking. As a result, system requirements can be easily specified using Bounded ACTL* and Temporal Specification Pattern, and effectively checked using existing action-based model checking framework and a toolchain.

As related works, There are process algebras and their tools providing model checking frameworks such as CCS [4] and CWB-NC [3]; CSP [5] and FDR [6]; FSP [7] and LTSA [7]. They are modeling languages for untimed system. However, [8] and [9] include a special action *tick* to model time in these languages, and provide timed model checking framework. There are extensions of temporal logic that allow one to model timed properties using bounded temporal operators [10], [11].

The rest of this paper is organized as follows. Sections 2 and 3 describes overviews of ACSR and model checking respectively. Section 4 explains our approach for ACSR model checking. Section 5 illustrates model checking of both timed and untimed properties on the Distance Control Module (DCM) as a case study. Finally, Sect. 6 concludes the paper.

2. Overview of ACSR

ACSR, like other process algebras, consists of (1) a set of operators and syntactic rules for constructing process; (2) a semantic mapping which assigns meaning or interpretation to processes; (3) a notion of equivalence or partial order between process; and (4) a set of algebraic laws that allows syntactic manipulation of processes. ACSR uses two distinct action types to model computation: time and resource-consuming actions, and instantaneous events.

2.1 The Computation Model

ACSR distinguish two types of actions: those which consume time, and those which are instantaneous. Timed ac-

Manuscript received April 16, 2009.
Manuscript revised July 3, 2009.

[†]The authors are with the Department of Computer Science and Engineering, Korea University, Korea.
[††]The author is with the Department of Computer and Information Science, University of Pennsylvania, U.S.A.
a) E-mail: choi@formal.korea.ac.kr

DOI: 10.1587/transfun.E92.A.2781

Copyright © 2009 The Institute of Electronics, Information and Communication Engineers

tions may require access to system resources, e.g., cpu's, memory, batteries, etc. In contrast, instantaneous actions provide a synchronization mechanism between concurrent processes.

Timed Actions. A system has a finite set of serially-reusable resources, $\mathcal{R}$. An action consumes one tick of time and employs a set of resources, each with an integer priority. For example, action $\{(r, p)\}$ denotes the use of some resource $r \in \mathcal{R}$ running at priority level p. The action $\emptyset$, consuming no resources, represents idling for one time unit.

Events. Instantaneous actions, or events, provide process synchronization in ACSR. An event is denoted by a pair (α, p), where α is the label of the event, and p is its priority. Labels represent input and output actions on channels. As in CCS, the special identity label, τ, arises when two events with input and output on the same channel synchronize. We define $\mathcal{L}$ as the set of all event labels.

We use $\mathcal{D}_R$ to denote the domain of timed actions, $\mathcal{D}_E$ to denote the domain of events, and $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$ to denote the entire domain of actions.

ACSR processes are described by the following grammar, where we assume a set of process constants each with an associated definition of the kind $C \stackrel{\text{def}}{=} P$.

$$P ::= NIL \mid (\alpha, n).P \mid A{:}P \mid P + P \mid P \| P \mid P\backslash F \mid [P]_I \mid P\backslash\backslash I \mid P\Delta_t^{\alpha}(P, P, P) \mid C$$

Steps of ACSR processes are constructed using the two prefix operators corresponding to the two types of actions. The process $(\alpha, n).P$ executes the instantaneous event (α, n) and proceeds to P. The process $A{:}P$ executes a resource-consuming action during the first time unit and proceeds to P. The process $P + Q$ represents nondeterministic choice and the process $P \| Q$ describes the concurrent composition of P and Q. The temporal scope construct, $P\Delta_t^{\alpha}(Q, R, S)$, restricts a process P by a time limit (t). If P completes its execution within this limit an exception, α, is thrown, in which case an exception handler (Q) is executed. If not, control is passed to a timeout process (R). In any case, P can be interrupted by a step of an interrupt process (S). Other static operators of ACSR allow us to hide the identity of certain resources $(P\backslash\backslash I)$, reserve the use of a resource for a given process $([P]_I)$, and force synchronization between processes by restricting certain events $(P\backslash F)$. The executions of a process are defined by a timed labeled transition system (timed LTS). A timed LTS, M, is defined as $(\mathcal{P}, \mathcal{D}, \rightarrow, P_0)$, where $\mathcal{P}$ is a set of ACSR processes, ranged over by P, Q, D is a set of actions, and $\rightarrow$ is a labeled transition relation such that $P \stackrel{\alpha}{\longrightarrow} Q$ if process P may perform an instantaneous event or timed action α and then behave as Q. $P_0 \in \mathcal{P}$ represents the initial state of the system.

The prioritized transition system is based on *preemption*, which incorporates our treatment of priority. This is based on a transitive, irreflexive, binary relation on actions, $\prec$ called the *preemption relation*. If $\alpha \prec \beta$, for two actions α and β, we say that α is *preempted by* β. Then, in any process, if there is a choice between executing either α or β, β will always be executed. We define the prioritized transition system "$\rightarrow_\pi$", which simply refines "$\rightarrow$" to account for preemption. The labeled transition system "$\rightarrow_\pi$" is defined as follows: $P \stackrel{\alpha}{\longrightarrow}_\pi P'$ if and only if (1) $P \stackrel{\alpha}{\longrightarrow} P'$ is an unprioritized transition, and (2) there is no unprioritized transition $P \stackrel{\beta}{\longrightarrow} P''$ such that $\alpha \prec \beta$. We refer to [12] for the precise definition of $\prec$ and semantics of ACSR operators.

2.2 Analysis of Real-Time Systems in ACSR

ACSR models can be analyzed in several ways. Similar to other behavioral formalisms, equivalence checking and model checking are common ways of establishing functional and timing correctness. In the former case, a detailed model is checked for equivalence with a more abstract model that represents system requirements. In the latter case, system requirements are expressed as formulas in a temporal logic and a model-checking algorithm is used to verify that the model satisfies these formulae.

Equivalence between ACSR processes is based on the concept of bisimulation [4], [13] which compares the computation trees of two processes. Two processes are bisimilar if, for each step of one, there is a matching, possibly multiple, step of the other, leading to bisimilar states. The formal definition of bisimulation for ACSR processes such as $\sim$, $\approx$, $\sim_\pi$ and $\approx_\pi$ can be found in [12]. Algorithms for checking strong and weak bisimulation for finite-state ACSR processes have been implemented in the VERSA toolset [14], thus allowing the verification of ACSR specifications.

ACSR models can be also checked against Hennessy-Milner logic(HML) formulas [2]. However, HML is not enough to describe requirements of a system in practice. In this paper, we suggest the use of ACTL* for analyzing ACSR models that will be explained in Sect. 3.

2.3 An Example

We describe an ACSR example, a Robot Control System (RCS). The RCS system consists of a binary semaphore and two user processes that run concurrently, and control robot's two arms. We assume the system should satisfy a safety property that the robot's two arms must not be operated at the same time in order to avoid power supply problem. Figure 1 shows expressions of ACSR processes of the system, and Fig. 2 depicts Timed LTSs corresponding to the ACSR processes.

We model a binary semaphore as a process *Sem* that receives events from user processes. When it receives *pend* in free state, it goes to non-free state. When it receives *post*

$$Sem \stackrel{\text{def}}{=} \emptyset : Sem + (pend, 0).rec\ X.(\emptyset : X + (post, 0).Sem)$$
$$P_1 \stackrel{\text{def}}{=} \emptyset : P_1 + (\overline{pend}, 1).(\overline{Ls}, 1).\{(left\_arm, 1)\} : (\overline{Le}, 1).(\overline{post}, 1).P_1$$
$$P_2 \stackrel{\text{def}}{=} \emptyset : P_2 + (\overline{pend}, 1).(\overline{Rs}, 1).\{(right\_arm, 1)\} : (\overline{Re}, 1).(\overline{post}, 1).P_2$$
$$RCS \stackrel{\text{def}}{=} (P_1 \| P_2 \| Sem)\backslash\{pend, post\}$$

Fig. 1 ACSR expressions of RCS.

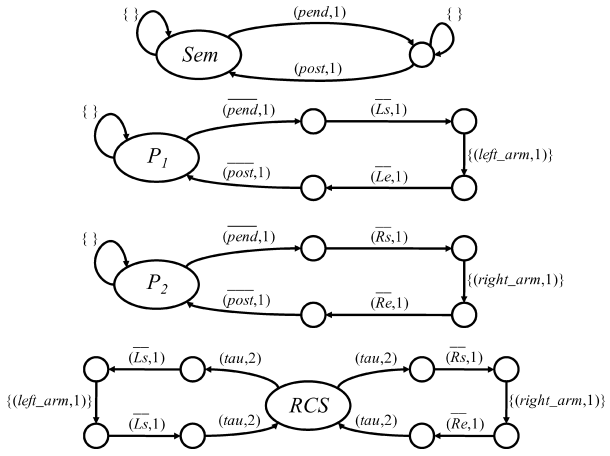

Fig. 2 Timed LTSs of RCS.

$$
\begin{aligned}
&Spec_1 \stackrel{\text{def}}{=} Spec_1' + Spec_1'' \\
&Spec_1' \stackrel{\text{def}}{=} (\tau,2).(\overline{Rs},1)\{(right\_arm,1)\} : (\overline{Re},1).(\tau,2).Spec_1 \\
&Spec_1'' \stackrel{\text{def}}{=} (\tau,2).(\overline{Ls},1)\{(left\_arm,1)\} : (\overline{Le},1).(\tau,2).Spec_1 \\
&Spec_2 \stackrel{\text{def}}{=} Spec_2' + Spec_2'' \\
&Spec_2' \stackrel{\text{def}}{=} (\overline{Rs},1)\{(right\_arm,1)\} : (\overline{Re},1).Spec_2 \\
&Spec_2'' \stackrel{\text{def}}{=} (\overline{Ls},1)\{(left\_arm,1)\} : (\overline{Le},1).Spec_2
\end{aligned}
$$

Fig. 3 Requirements specifications of RCS.

in non-free state, it goes to free state.

We model the user processes as the processes $P1$ and $P2$ that operate robot's left and right arm respectively. To satisfy the safety property, the processes $P1$ and $P2$ should arbitrate their execution using the semaphore. $P1$ and $P2$ send the event *pend* to acquire the semaphore, and the event *post* to release it. When $P1$ or $P2$ starts using left or right arm, it marks the start of this activity by the event *Ls* or *Rs* respectively. When $P1$ or $P2$ stops operating the left or right arm, it marks the end of this activity by the event *Le* or *Re* respectively.

Figure 3 shows two requirements specifications for the system. $Spec_1$ describes all possible actions including synchronized events which are represented by the internal event τ(tau), while $Spec_2$ abstracts out the internal events. In both requirements, one can see that the two resources *left_arm* and *right_arm* aren't used at the same time. We can prove that

$$RCS \sim_\pi Spec_1 \text{ and}$$
$$RCS \approx_\pi Spec_2.$$

The RCS system can be verified by modeling its requirement specification and equivalence checking. However, for large and complex systems, it is hard to model requirement specification using ACSR because it usually involves sequentializing the behaviors of parallel processes to inspect that the specification satisfies all requirements of the system, and it doesn't contain undesirable behaviors. In this paper, we suggest the use of model checking for verifying real-time properties of ACSR models.

3. Overview of Model Checking

Model checking is an automated technique for checking finite-state reactive systems against properties described by temporal logics [15]. The method has been used to verify software and hardware designs of complex systems. A system is modeled into a system model and requirements of the system are formalized into a property specification. Model checking technique can check whether the system model satisfies the property specification and provides counterexample if it doesn't satisfy.

3.1 State-Based vs. Action-Based Modeling

A system model can be either state-based, action-based, or based on their mixture [16], [17]. ACSR models are action-based. We explain the difference between state-based and action-based modeling.

In state-based modeling, a state of a system model is a valuation of variables composing the system model. Kripke Structures (KSs) are used for representing state-based models. CTL* [18], [19], CTL [20] and LTL [21] describe properties of state-based models. Atomic propositions of the temporal logics are predicates over state variables of the system model. For example, model description languages such as SMV Language [22] and Promela [23] are the input languages of model checkers NuSMV [22] and SPIN [23] respectably, which are state-based.

In action-based modeling, system models are represented by Labeled Transition Systems (LTSs) rather than Kripke Structures. Labels are on states in KSs, but labels are on transitions in LTSs. In LTSs, labels are actions and the actions are used as atomic propositions in action-based temporal logics. The temporal logics such as Action CTL*(ACTL*) [16], Action CTL(ACTL) [16] and Action LTL(ALTL) are used for describing properties of action-based models. Atomic propositions of the temporal logics are predicates over actions of the system model. For example, many process algebras including ACSR and CCS interpreted by LTSs are action-based modeling languages.

3.2 Action-Based Model Checking

Action-based model checking uses LTSs as system models and action-based temporal logics for specifying system properties. We define LTSs and ACTL*.

A Labeled Transition System M is a quadruple $(\mathcal{P}, \mathcal{D}, \rightarrow, P_0)$ where

- $\mathcal{P}$ is a set of states,
- $\mathcal{D}$ is a set of actions,
- $\rightarrow \subseteq \mathcal{P} \times \mathcal{D} \times \mathcal{P}$ is a transition relation, and
- $P_0 \in \mathcal{P}$ is an initial state.

An execution $\chi = (P_1, \alpha_1, P_2)(P_2, \alpha_2, P_3)\ldots \in \rightarrow^\omega$

from P_1 is an infinite sequence of transitions. χ_2 is a suffix of χ_1, denoted by $\chi_1 \leq \chi_2$ if there is an execution χ such that $\chi_1 = \chi \cdot \chi_2$ (where the operation '$\cdot$' is concatenation). χ_2 is a proper suffix of χ_1, denoted by $\chi_1 < \chi_2$ if $\chi_1 \leq \chi_2$ and $\chi_1 \neq \chi_2$.

We define ACTL*. The set of atomic proposition AP is the set of actions A. State and path formulas are defined with path quantifier **E**; temporal operators **X** and **U** as follows:

- If ψ_1 and ψ_2 are state formulas, then $\neg\psi_1$ and $\psi_1 \vee \psi_2$ are state formulas.
- If ϕ is a path formula, then $\mathbf{E}\phi$ is a state formula.
- If $\alpha \in$AP, then α is a path formula.
- If ψ is a state formula, then ψ is also a path formula.
- If ϕ_1 and ϕ_2 are path formulas, then $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\mathbf{X}\phi_1$ and $\phi_1\mathbf{U}\phi_2$ are path formulas.

An interpretation for a state formula is a pair of a LTS M and a state P. An interpretation for a path formula is a pair of a LTS M and a path χ. The semantics of the temporal logic formulas is defined as follows:

- $M, P \models \neg\psi$ iff $M, P \not\models \psi$.
- $M, P \models \psi_1 \vee \psi_2$ iff $(M, P \models \psi_1)$ or $(M, P \models \psi_2)$.
- $M, P \models \mathbf{E}\phi$ iff there is a path χ from P such that $M, \chi \models \phi$.
- $M, \chi \models \psi$ iff P is the first state of χ, and $M, P \models \psi$.
- $M, \chi \models \alpha$ iff α is the action of the first transition in χ.
- $M, \chi \models \neg\phi$ iff $M, \chi \not\models \phi$.
- $M, \chi \models \phi_1 \vee \phi_2$ iff $(M, \chi \models \phi_1)$ or $(M, \chi \models \phi_2)$.
- $M, \chi \models \mathbf{X}\phi$ iff there exists P_1, α, P_2 and χ' such that $\chi = (P_1, \alpha, P_2) \cdot \chi'$ and $M, \chi' \models \phi$.
- $M, \chi \models \phi_1\mathbf{U}\phi_2$ iff there exists $\chi_2 \geq \chi$ such that $M, \chi_2 \models \phi_2$ and for all $\chi \leq \chi_1 < \chi_2 : M, \chi_1 \models \phi_2$.

Auxiliary notations for ACTL* are defined as follows: true $\equiv \phi \vee \neg\phi$, false $\equiv \neg$true, $\phi_1 \wedge \phi_2 \equiv \neg(\neg\phi_1 \vee \neg\phi_2)$, $\psi_1 \wedge \psi_2 \equiv \neg(\neg\psi_1 \vee \neg\psi_2)$, $\phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$, $\psi_1 \rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2$, $\mathbf{F}\ \phi \equiv$ true $\mathbf{U}\ \phi$, $\mathbf{G}\ \phi \equiv \neg\ \mathbf{F}\ \neg\phi$, and $\phi_1\ \mathbf{W}\ \phi_2 \equiv (\phi_1\ \mathbf{U}\ \phi_2) \vee \mathbf{G}\ \phi_1$.

We use this action based model checking framework. However, it is not suitable for timed model checking. We extends this framework in the next section.

4. Model Checking of ACSR Models

ACSR models are action-based and represented by TLTSs. A TLTS can be either unprioritized or prioritized. In model checking for ACSR, we consider only prioritized TLTSs, so we refer to 'prioritized TLSTs' as just 'TLTSs' from now on. Given a TLTS M and an ACTL* formula ψ, ACSR model checking problem can be defined as to check whether $M \models \psi$ (iff $M, P_0 \models \psi$).

Formulas of the logic will be interpreted over TLTSs generated by ACSR processes. Formulas, therefore, will refer to the labels of the transition systems, that is, instantaneous events and timed actions. However, events and actions carry with them the values of their dynamic attributes, which are not meaningful in the logical context. In our model checking framework, we focus on temporal properties in terms of occurrence of observable events and passage of time rather than usage of resources. We abstract every timed action to the special action *tick*. Therefore, primitive constructs used in the logical formulas are event labels, and *tick*, as action labels. Given an event $e = (a, p)$, we write $abs(e) = a$ and, given a timed action A, we write $abs(A) = tick$. We use $\mathcal{A}$ to denote the domain of abstracted action such that $\mathcal{A} = \mathcal{L} \cup \{tick\}$. For model checking of ACSR models, we use $\mathcal{A}$ as the set of atomic propositions of ACTL*. We refer to 'ACTL* over $\mathcal{A}$' simply as 'ACTL*' from now on.

ACSR model checking problem is to check whether $M \models \psi$ where M is a prioritized TLTS, and ψ is an ACTL* formula over $\mathcal{A}$. The semantics of ACTL* interpreted by a TLTS M are almost same with those in Sect. 3.2 except:

- $M, \chi \models a$ iff for some p, the action of the first transition in χ is (a, p) where $a \in \mathcal{L}$.
- $M, \chi \models tick$ iff the action of the first transition in χ ranges over $\mathcal{D}_R$.

For example, there is a requirement for the RCS system in Sect. 2.3, that is, "Every task that involves using the one of the arms should end at some time." This is a non real-time property and can be specified as $\mathbf{AG}(\overline{Ls} \rightarrow \mathbf{F}\ \overline{Le})$ and $\mathbf{AG}(\overline{Rs} \rightarrow \mathbf{F}\ \overline{Re})$.

To provide timed model checking for ACSR, we, therefore, (1) translate TLTSs into Abstracted TLTSs where resources of timed actions and priorities of instantaneous actions are abstracted and removed while preserving satisfiability for ACTL*, and (2) extend ACTL* to Bounded ACTL* by adding bounded temporal operators in order to easily describe properties with respect to time.

To easily specify requirements on a system, we use Temporal Specification Pattern [24], [25] that provide the category of specification formulas which frequently used in practice. To realize ACSR model checking, we use the toolchain composed by (1) VERSA that can generate state space of ACSR models (TLTSs), abstract them into ATLTS, and translate Bounded ACTL* and specification patterns to ACTL*; and (2) CWB-NC that can check ATLTSs against ACTL* formulas.

4.1 Translating TLTSs to Abstracted TLTSs

We define Abstracted TLTSs (ATLTS) as LTSs with the domain of abstracted actions $\mathcal{A}$. A TLTS $M = (\mathcal{P}, \mathcal{D}, \rightarrow, P_0)$ is translated to an ATLTS $M = (\mathcal{P}', \mathcal{A}, \rightarrow', P_0')$ where

- $\mathcal{P}' = \mathcal{P}$
- $\rightarrow' = \{(P, abs(\alpha), P') | (P, \alpha, P') \in \rightarrow\}$
- $\mathcal{A} = \{abs(\alpha) | \alpha \in \mathcal{D}\}$
- $P_0' = P_0$

This translation preserves satisfiability for ACTL*. We can prove that for all ψ, $M \models \psi$ iff $M' \models \psi$ where M is a TLTS and M is translated to a ATLTS M'. This can be proved by contradiction. If there exists ψ such that $M \not\models$

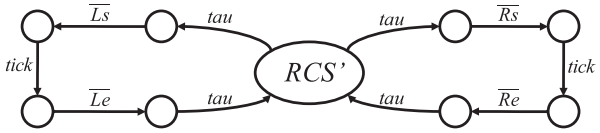

Fig. 4 ATLTS of RCS.

ψ and $M' \models \psi$, then the counterexample of M can easily produce an execution of M' which falsifies $M' \models \psi$, and viceversa.

ACSR are used for modeling real-time systems, so it needs timed model checking to verify the timed systems. TLTSs have two types of actions. One is the type of instantaneous actions and another is the type of timed actions. We abstract timed actions of TLTSs into the *tick* action of ATLTS. By translating TLTS to ATLTS, time-consuming is limited to only on *tick* actions; it makes timed model checking for ACSR simple. Without this abstraction, specifying passage of time units in ACTL* becomes very difficult because it involves referring all timed actions for describing passage of one time unit. An ATLTS is an resource- and priority-abstraction of an TLTS, and ATLTS are still models of discrete timed system. The time of the ATLTS in a given state is counted by the number of *tick* actions that have occurred since the beginning of the execution.

When timed actions are translated to *tick* actions, information on resources and priorities are removed. This information is essential for the semantics of ACSR operators, such as parallel composition and prioritization. We, however, model check the final ACSR model where all parallel compositions and prioritization has finished. We analyze a sequence of observable instantaneous actions (events) of ACSR models rather than resources or priorities. Therefore, information on resources and priorities can be removed. Priorities on instantaneous actions are abstracted and removed for the same reason.

For example, Fig. 4 shows the ATLTS translated from the TLTS *RCS* depicted in Fig. 2.

4.2 Extending ACTL* to Bounded ACTL*

ACTL* is not suitable for describing timed properties. In order to specify requirements on timed model, we extend ACTL* to Bounded ACTL* by adding bounded temporal operators. Bounded temporal operators come from Metric Temporal Logic [26]. We add bounded path formulas which composed by bounded temporal operators as follows:

- If ϕ_1 and ϕ_2 are path formulas, then $(\mathbf{F}^{\sim d}\phi_1)$, $(\mathbf{G}^{\sim d}\phi_1)$ and $(\phi_1 \mathbf{U}^{\sim d}\phi_2)$ are path formulas where $\sim \in \{ <, >, \leq, \geq \}$ and d is a natural number.

Bounded path formulas can assert properties of system models with respect to time. For example, $\mathbf{F}^{\leq d}$ is a bounded temporal operator which is extended from the unbounded temporal operator $\mathbf{F}$. $\mathbf{F}^{\leq d}\phi$ means ϕ holds at some future time within the next d time units. In terms of the action *tick*, The meaning of $\mathbf{F}^{\leq d}\phi$ is that ϕ holds in the future before next $d+1$ *tick* actions occur.

The semantics of the bounded path formulas is defined over the temporal distance function *dist* as follows:

- $M,\chi \models \mathbf{G}^{\sim d}\phi$ iff $M,\chi_1 \models \phi$ for all $\chi_1 \geq \chi$ and $dist(\chi, \chi_1) \sim d$.
- $M,\chi \models \mathbf{F}^{\sim d}\phi$ iff $M,\chi_1 \models \phi$ for some $\chi_1 \geq \chi$ and $dist(\chi, \chi_1) \sim d$.
- $M,\chi \models \phi_1 \mathbf{U}^{\sim d}\phi_2$ iff $M,\chi_1 \models \phi_2$ for some $\chi_1 \geq \chi$ and $dist(\chi, \chi_1) \sim d$ and $M,\chi_2 \models \phi_1$ for all χ_2 such that $\chi \leq \chi_2 < \chi_1$

where $dist(\chi_1, \chi_2)$ is the number of path χ such that $\chi_1 \leq \chi < \chi_2$ and the first action of χ is *tick*.

It is possible to translate bounded path formulas to unbounded path formula with their semantics preserved. The translation function *tr* that comes from [9] is defined as follows:

- $tr(\mathbf{G}^{<d}\phi) = \begin{cases} \phi \mathbf{W} (tick \wedge \phi), & \text{if } d = 1; \\ \phi \mathbf{W} (tick \wedge \phi \wedge tr(\mathbf{G}^{<d-1}\phi)), & \text{if } d > 1. \end{cases}$
- $tr(\mathbf{G}^{\leq d}\phi) = tr(\mathbf{G}^{<d+1}\phi)$
- $tr(\mathbf{F}^{<d}\phi) = \begin{cases} \neg tick \mathbf{W} \phi, & \text{if } d = 1; \\ (\neg tick \vee \mathbf{X}\, tr(\mathbf{F}^{<d-1}\ \phi)) \mathbf{W} \phi, & \text{if } d > 1. \end{cases}$
- $tr(\mathbf{F}^{\leq d}\phi) = tr(\mathbf{F}^{<d+1}\phi)$.
- $tr(\mathbf{G}^{\geq d}\phi) = \begin{cases} \mathbf{G}\, \phi, & \text{if } d = 0; \\ \neg tick \mathbf{W} \mathbf{G}\, \phi, & \text{if } d = 1; \\ \neg tick \mathbf{W} \mathbf{X}\, tr(\mathbf{G}^{\geq d-1}\ \phi), & \text{if } d > 1. \end{cases}$
- $tr(\mathbf{G}^{>d}\phi) = tr(\mathbf{G}^{\geq d+1}\phi)$.
- $tr(\mathbf{F}^{\geq d}\phi) = \begin{cases} \mathbf{F}\, \phi, & \text{if } d = 0; \\ \mathbf{F}\, (tick \wedge \mathbf{F}\, \phi), & \text{if } d = 1; \\ \mathbf{F}\, (tick \wedge \mathbf{X}\, tr(\mathbf{F}^{\geq d-1}\ \phi)), & \text{if } d > 1. \end{cases}$
- $tr(\mathbf{F}^{>d}\phi) = tr(\mathbf{F}^{\geq d+1}\phi)$.
- $tr(\phi_1 \mathbf{U}^{\sim d}\phi_2) = tr(\mathbf{F}^{\sim d}\phi_2) \wedge (\phi_1 \mathbf{W} \phi_2)$

where $\mathbf{W}$ is the unbounded weak-until operator introduced in Sect. 3.2.

For example, there is a real-time requirement for the RCS system, that is, "Every task that involves using the one of the arms should end at some future time within one time unit." This real-time property can be specified as $\mathbf{AG}(\overline{Ls} \rightarrow \mathbf{F}^{\leq 1}\ \overline{Le})$ and $\mathbf{AG}(\overline{Rs} \rightarrow \mathbf{F}^{\leq 1}\ \overline{Re})$.

4.3 Temporal Specification Pattern

Temporal specification pattern is a general reusable solution to a occurring problem in specifying a system's requirements using temporal logics. [24] provides untimed temporal specification patterns and [25] provides more patterns for real-time system. We apply the patterns to our case study in Sect. 5.

For example, in the RCS system robot should not start moving left arm when it is moving the right arm. In other words, the event *Le* 'never' occur 'between' the events *Rs* and *Re*. This requirement of natural language is translated to ACTL* formula using Absence(Never) and Between patterns [24] as:

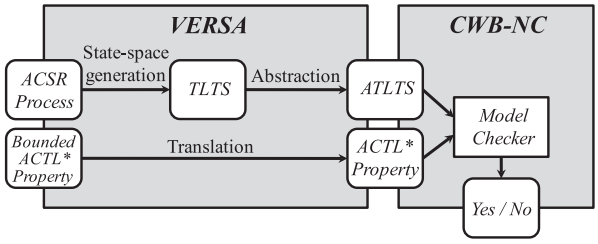

Fig. 5 Tool chain.

$$\mathbf{AG}(\overline{Rs} \rightarrow (\neg\overline{Ls}\ \mathbf{U}\ \overline{Re}))$$

[25] provides real-time specification patterns that could be used to specify requirements of ACSR models. Some examples of this are presented in Sect. 5.

4.4 Tool Chain

VERSA [14] is a tool for ACSR. VERSA can verify, execute and rewrite ACSR processes. CWB-NC [3] is a verification environment for process algebras including CCS. CWB-NC takes a LTS as an input, and supports model checking of the LTS against ACTL* formulas.

We use a toolchain composed by VERSA and CWB-NC. Figure 5 shows the architecture of our toolchain. VERSA generate a state space for an given ACSR specification and translate it to an ATLTS. Properties for ACSR models are described by Bounded ACTL* and Temporal Specification Patterns. The temporal properties are translated to unbounded ACTL* by VERSA. Given an ATLTS and an ACTL* property, CWB-NC check the models against the property.

5. Case Study

We applied ACSR model checking to an example from industry. In this paper, we present Distance Control Module (DCM), a subsystem of a new railroad signaling system that is under research in Korea Railroad Research Institute (KRRI).

5.1 DCM System Description

The Distance Control Module (DCM) system is a subsystem of a railroad block signaling system which has a concept of both fixed and moving block interlocking. The DCM system manages 'running trains' by calculating and sending permission for the trains to enter the next blocks and limiting temporary speed of the trains, so that the trains can keep a safe distance to obstacles and other trains ahead. The permission is called *Permissive Movement Authority* (PMA). A PMA can be Green, Yellow, or Red that represents the level of warning for obstacles ahead. Trains have to manage their speeds according to the color of PMA that they receive.

The functions of the DCM system are (1) controlling distance between a train and an obstacle ahead, (2) train positioning, (3) monitoring train direction, (4) limiting temporary train speed, and (5) opening and closing blocks. In this paper, we present ACSR models of an abstracted subset of DCM to demonstrate the usefulness of ACSR model checking.

$$
\begin{aligned}
TP\_ready &\stackrel{\text{def}}{=} (dispatch\_TP, 1).TP\_running + \emptyset : TP\_ready\\
TP\_wait &\stackrel{\text{def}}{=} (timeTick, 2).TP\_ready + \emptyset : TP\_wait\\
TP\_running &\stackrel{\text{def}}{=} (\overline{TPs}, 1).\{(cpu, 1)\} : (\{(cpu, 1)\} : TP\_running' +\\
&\qquad (\overline{ActUP}, 1).\{(cpu, 1)\} : (\overline{post}, 1).TP\_running')\\
TP\_running' &\stackrel{\text{def}}{=} (\overline{TPe}, 1).(\overline{schdl}, 1).TP\_wait\\
\\
UP\_ready &\stackrel{\text{def}}{=} (dispatch\_UP, 1).UP\_running + \emptyset : UP\_ready\\
UP\_wait &\stackrel{\text{def}}{=} (\overline{pend}, 1).UP\_ready + \emptyset : UP\_wait\\
UP\_running &\stackrel{\text{def}}{=} (\overline{UPs}, 1).\{(cpu, 1)\}^2 :\\
&\qquad (\{(cpu, 1)\} : UP\_running' + \{(cpu, 1)\}^2 : UP\_running')\\
UP\_running' &\stackrel{\text{def}}{=} (\overline{UPe}, 1).(\overline{schdl}, 1).UP\_wait\\
\\
SP\_ready &\stackrel{\text{def}}{=} (dispatch\_SP, 1).SP\_running + \emptyset : SP\_ready\\
SP\_wait &\stackrel{\text{def}}{=} (timeTick, 1).SP\_ready + \emptyset : SP\_wait\\
SP\_running &\stackrel{\text{def}}{=} (\overline{SPs}, 1).(\{(cpu, 1)\} : SP\_running' + \{(cpu, 1)\}^2 : SP\_running')\\
SP\_running' &\stackrel{\text{def}}{=} (\overline{SPe}, 1).(\overline{schdl}, 1).SP\_wait\\
\\
Sem\_free &\stackrel{\text{def}}{=} (pend, 1).Sem\_nonfree + \emptyset : Sem\_free\\
Sem\_nonfree &\stackrel{\text{def}}{=} (post, 1).Sem\_free + \emptyset : Sem\_nonfree\\
\\
Timer &\stackrel{\text{def}}{=} \emptyset^{10} : (\overline{TimeTick}, 1).(\overline{timeTick}, 1).(\overline{timeTick}, 1).Timer\\
\\
Scheduler &\stackrel{\text{def}}{=} (\overline{dispatch\_TP}, 2).Scheduler' + (\overline{dispatch\_UP}, 3).Scheduler' +\\
&\qquad (\overline{dispatch\_SP}, 1).Scheduler' + \emptyset : Scheduler\\
Scheduler' &\stackrel{\text{def}}{=} (schdl, 1).Scheduler + \emptyset : Scheduler'\\
\\
DCM &\stackrel{\text{def}}{=} (TP\_ready||UP\_wait||SP\_ready||Sem\_nonfree||Timer||Scheduler)\backslash\\
&\qquad \{timeTick, pend, post, dispatch\_TP, dispatch\_UP, dispatch\_SP, schdl\}
\end{aligned}
$$

Fig. 6 ACSR models of DCM.

5.2 ACSR Models of DCM

DCM is a real-time system based on a real-time operating system. We model some components of a real-time operating system (RTOS) such as *Semaphore (Sem)*, *Timer*, and *Scheduler* in Fig. 6. We model user tasks that perform the DCM functions as ACSR processes *Train Positioning (TP)*, *Updating PMAs (UP)* and *Sending PMAs (SP)*. *TP* receives positions, directions and speeds of trains periodically. *TP* activates *UP* if a calculation for updating PMAs is needed. *UP* recalculates PMAs. *SP* sends PMAs to the trains periodically. This model is verified against real-time properties with respect to schedulerability issues by model checking.

In our RTOS model, a task state can be *ready*, *wait*, or *running*. The tasks *TP* and *SP* are periodic tasks, which are executed in each scheduling period. The length of scheduling period is 10 time units. The execution time of *TP* is 2 time units. The execution time of *SP* is 2 or 3 time units. *UP* is an aperiodic task awaken by *TP* through semaphore mechanism. *TP* is modeled, for every scheduling period, to decide whether or not to awake *UP* in wait state nondeterministically. *TPs*, *UPs* and *SPs* are observable events for marking the start of execution of a task. *TPe*, *UPe* and *SPe* are observable events for marking the end of execution of a task. In Fig. 7 shows possible execution order of tasks. For each scheduling period, *TP* is executed firstly, and *SP* is executed lastly. If *TP* activates *UP*, *UP* is executed between

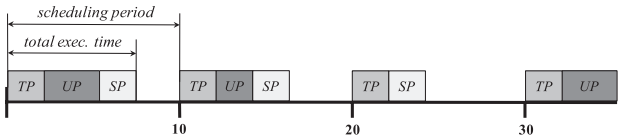

Fig. 7 Architecture of the miniature model.

TP and *SP*.

In our model, we use a semaphore for synchronizing *TP* and *UP*. The semaphore state is either *free* or *nonfree*. When it receives *pend* in free state, it goes to non-free state. When it receives *post* in nonfree state, it goes to free state.

Timer is used for periodical scheduling. Every 10 time units, *Timer* make $\overline{TimeTick}$ occur which indicates the beginning of a new scheduling period, and also make *timeTick* occur to awake the tasks *TP* and *SP*.

Scheduler dispatches one of the tasks in ready state which has the highest priority. The priorities of tasks *TP*, *UP*, and *SP* are 2, 3, and 1, respectively. For each scheduling period, *TP* is executed firstly, and *SP* is executed lastly.

5.3 Model Checking of DCM

The DCM system has requirements in natural language as follows:

- **R1** *TP* is executed every scheduling period ($\overline{TPs}$ action occurs between two consecutive $\overline{TimeTick}$ actions).
- **R2** *SP* is executed every scheduling period ($\overline{SPs}$ action occurs between two consecutive $\overline{TimeTick}$ action).
- **R3** *TP* finish its execution after at most 2 time units from the beginning of every scheduling period (If $\overline{TimeTick}$ occurs, then $\overline{TPe}$ occurs after at most 2 time units).
- **R4** *SP* is the last task executed for every scheduling period (There is no $\overline{TPs}$ nor $\overline{UPs}$ action between $\overline{SPe}$ and $\overline{TimeTick}$ events).
- **R5** The CPU utilization is no greater than 70% (i.e., *SP* finish its execution after at most 7 time units from the beginning of every scheduling period) (If $\overline{TimeTick}$ occurs, then $\overline{SPe}$ occurs after at most 7 time units).
- **R6** There can be a period such that the total execution time is 3 in it (The earliest end time of *SP* in some period is 3).
- **R7** When *TP* finish its execution, if *TP* decides to activate *UP*, then *UP* will starts its execution with no time delay (If $\overline{ActUP}$ occurs, then $\overline{UPs}$ occurs with no time delay).

We formulate the requirements in bounded ACTL*. The requirements above can be translated to temporal logic formulas in terms of observable events.

- **R1** $\mathbf{AG}(\overline{TimeTick} \rightarrow \mathbf{X}\,(\neg\overline{TimeTick}\ \mathbf{W}\ \overline{TPs}))$
- **R2** $\mathbf{AG}(\overline{TimeTick} \rightarrow \mathbf{X}\,(\neg\overline{TimeTick}\ \mathbf{W}\ \overline{SPs}))$
- **R3** $\mathbf{AG}(\overline{TimeTick} \rightarrow \mathbf{F}^{\leq 2}\overline{TPe})$
- **R4** $\mathbf{AG}(\overline{SPe} \rightarrow (\neg(\overline{TPs} \vee \overline{UPs})\ \mathbf{W}\ \overline{TimeTick})$
- **R5** $\mathbf{AG}(\overline{TimeTick} \rightarrow \mathbf{F}^{\leq 7}\overline{SPe})$
- **R6** $\mathbf{EF}(\overline{TimeTick} \wedge \mathbf{F}^{\leq 3}\overline{SPe})$
- **R7** $\mathbf{AG}(\overline{ActUP} \rightarrow \mathbf{F}^{\leq 0}\overline{UPs})$

Table 1 Comparative result of the case study.

Analysis Method	Expressiveness	Clearness	Available Pattern	Tool Support	Time
Bounded ACTL*	O	O	O	VERSA - CWB tool chain	< 2 sec
HMLu	O	X	X	None	N.A.
Bisimulation	X	N.A.	X	VERSA	N.A

While the requirements **R1**, **R2** and **R4** are untimed, the requirements **R3**, **R5**, **R6** and **R7** are timed properties specified as bounded temporal formulas. The bounded formulas **R3**, **R5**, **R6** and **R7** are translated to unbounded formulas. For example, **R3** is translated to the unbounded formula, $\mathbf{AG}(\overline{TimeTick} \rightarrow ((\neg tick \vee \mathbf{X}\,((\neg tick \vee \mathbf{X}\,(\neg tick\ \mathbf{W}\ \overline{TPe}))\ \mathbf{W}\ \overline{TPe}))\ \mathbf{W}\ \overline{TPe}))$.

We use untimed patterns of [24] and timed patterns of [25]. For untimed properties, we apply the Existence/Between pattern to **R1** and **R2**, and the Absence/Between pattern to **R4**. The timed properties **R3**, **R5**, **R6** and **R7** are specified using the Bounded Response pattern.

All properties are checked against the DCM model by the toolchain of VERSA and CWB-NC. VERSA translates the DCM model to a TLTS, and abstracts the TLTS into an ATLTS. Each of the TLTS and ATLTS has 55 states and 62 transitions. The bounded formulas are translated to unbounded ones by VERSA. It takes less than two seconds for checking all the properties in CWB-NC.

5.4 Discussion

Table 1 shows comparative results of the case study among the analysis methods of ACSR such as (1) bounded ACTL* model checking that we suggest in this paper, (2) Extended HML with Until (HMLu) model checking [2], and (3) bisimulation checking.

In bounded ACTL* model checking, bounded ACTL* is so expressive that all the requirements are specified as ACTL* formulas in Sect. 5.3. The formulas are clear to read so that one can investigate whether they really reflect the natural language requirements. In addition, temporal specification patterns are available for bounded ACTL* so that one can reduce mistakes in writing specifications. All the requirements are checked within two seconds by support of the toolchain composed by VERSA and CWB-NC.

The requirements in the case study can be specified using HMLu with difficulty. HMLu is less expressive than bounded ACTL*. The HMLu formulas are too complicated to read and write in practice. For example, the HMLu formula corresponding to **R1** is $\neg(((tt < TimeTick > tt) \wedge (\neg tt < \mathcal{A} > ((tt < TimeTick > tt) < \mathcal{A}* > \neg(tt < TPs > tt)))) < \mathcal{A}* > tt)$ where $\mathcal{A}$ is the domain of abstracted actions. The temporal specification patterns are unavailable for HMLu. In addition, there is no efficient algorithm and tool support for HMLu model checking.

To analyze ACSR models using bisimulation checking, specifications of the models should be specified as ACSR processes. It is almost impossible that each requirement from **R1** to **R7** is separately represented by an ACSR process. It is also unmanageable to write specification models having all the behaviors the requirements demand and the other desirable behaviors. Moreover, there is no available pattern to write specifications for bisimulation checking.

Bounded ACTL* model checking is an useful approach to analyze ACSR models. ACSR can be used as a fundamental theory giving formal semantics to higher-level design or specification languages [27], [28]. A sample application domain of ACSR is analysis of scheduling problems in resource-bound real-time embedded system development [29], [30]. The other domains are protocol analysis and rapid prototyping. Bounded ACTL* model checking will play an important role in analyzing ACSR models in the domains.

6. Conclusion and Future Work

We presented an framework to provide timed model checking for models of concurrent real-time system expressed in ACSR. The framework works by translating a TLTS to ATLTS, extending ACTL* to Bounded ACTL*, and using a toolchain composed by VERSA and CWB-NC. We also presented analysis of the DCM system within the proposed ACSR model checking framework. As a result, system requirements could be easily specified using Bounded ACTL* and temporal specification patterns, and effectively checked using our toolchain.

Future work is to extend this framework to allow one to specify and verify resource related requirements using ACSR and model checking technique. ACSR employs resources as a basic primitive. A real-time system depends not only on delays due to process synchronization, but also on the availability of shared resources. Model checking of resource related properties in ACSR is the subject for a future study.

Acknowledgements

This research was partially supported by Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/Korea Science and Engineering Foundation(KOSEF), grant number R11-2008-007-03002-0, and Korea SW Industry Promotion Agency (KIPA) under the program of Software Engineering Technologies Development and Experts Education.

References

[1] I. Lee and R. Gerber, "A process algebraic approach to the specification and analysis of resource-bound real-time systems," Proc. IEEE on Real-Time Systems, pp.158–171, 1994.

[2] I. Lee, A. Philippou, and O. Sokolsky, "Resources in process algebra," J. Logic and Algebraic Programming, vol.72, no.1, pp.98–122, 2007. Algebraic Process Calculi: The First Twenty Five Years and Beyond. II.

[3] R. Cleaveland, J. Parrow, and B. Steffen, "The concurrency workbench: A semantics-based tool for the verification of concurrent systems," ACM Trans. Program. Lang. Syst., vol.15, no.1, pp.36–72, 1993.

[4] R. Milner, Communication and concurrency, Prentice-Hall, Upper Saddle River, NJ, USA, 1989.

[5] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International, 1985.

[6] A.W. Roscoe, C.A.R. Hoare, and R. Bird, The Theory and Practice of Concurrency, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[7] J. Magee and J. Kramer, Concurrency: State Models & Java Programs, John Wiley & Sons, New York, NY, USA, 1999.

[8] M.J. Morley, "Safety-level communication in railway interlockings," Sci. Comput. Program., vol.29, no.1-2, pp.147–170, 1997.

[9] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Fluent temporal logic for discrete-time event-based models," SIGSOFT Softw. Eng. Notes, vol.30, no.5, pp.70–79, 2005.

[10] R. Koymans, Specifying message passing and time-critical systems with temporal logic, Springer-Verlag New York, Secaucus, NJ, USA, 1992.

[11] T.A. Henzinger, "It's about time: Real-time logics reviewed," Proc. 9th International Conference on Concurrency Theory, pp.439–454, 1998.

[12] P. Brémond-Grégoire, J.Y. Choi, and I. Lee, "A complete axiomatization of finite-state ACSR processes," Inf. Comput., vol.138, no.2, pp.124–159, 1997.

[13] D. Park, "Concurrency and automata on infinite sequences," Proc. 5th GI-Conference on Theoretical Computer Science, pp.167–183, Springer-Verlag, London, UK, 1981.

[14] D. Clarke, I. Lee, and H.L. Xie, "Versa: A tool for the specification and analysis of resource-bound real-time systems," J. Comput. Soft. Eng., vol.3, no.2, pp.189–215, 1995.

[15] E.M. Clarke, Jr., O. Grumberg, and D.A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.

[16] R. De Nicola and F. Vaandrager, "Action versus state based logics for transition systems," Proc. LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes, pp.407–419, Springer-Verlag New York, New York, NY, USA, 1990.

[17] S. Chaki, E.M. Clarke, N. Sharygina, and N. Sinha, "State/event-based software model checking," in In Integrated Formal Methods, pp.128–147, Springer-Verlag, 2004.

[18] E.M. Clarke and E.A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," Proc. IBM Workshop on Logic of Programs, pp.52–71, 1982.

[19] L. Lamport, ""Sometime" is sometimes "not never": On the temporal logic of programs," Proc. 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80), pp.174–185, 1980.

[20] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Trans. Program. Lang. Syst., vol.8, no.2, pp.244–263, 1986.

[21] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag New York, New York, NY, USA, 1992.

[22] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model verifier," CAV'99: Proc. 11th International Conference on Computer Aided Verification, pp.495–499, Springer-Verlag, London, UK, 1999.

[23] G. Holzmann, "The model checker spin," IEEE Trans. Softw. Eng., vol.23, no.5, pp.279–295, May 1997.

[24] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, "Patterns in property specifications for finite-state verification," ICSE'99: Proc. 21st International Conference on Software Engineering, pp.411–420, ACM, New York, NY, USA, 1999.

[25] S. Konrad and B.H. Cheng, "Real-time specification patterns,"

ICSE'05: Proc. 27th International Conference on Software Engineering, pp.372–381, ACM, New York, NY, USA, 2005.
[26] R. Alur and T.A. Henzinger, "Real-time logics: Complexity and expressiveness," Inf. Comput., vol.104, no.1, pp.35–77, 1993.
[27] O. Sokolsky, I. Lee, and D. Clarke, "Schedulability analysis of aadl models," 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, p.8, April 2006.
[28] S. Fischmeister, O. Sokolsky, and I. Lee, "Network-code machine: Programmable real-time communication schedules," IEEE Real-Time and Embedded Technology and Applications Symposium, pp.311–324, 2006.
[29] I. Lee, A. Philippou, and O. Sokolsky, "A general resource framework for real-time systems," in Radical Innovations of Software and Systems Engineering in the Future, ScholarlyCommons@Penn, 2002.
[30] O. Sokolsky, "Resource modeling for embedded systems design," IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, p.99, 2004.

Junkil Park received the B.S. degree in Computer Science and Engineering from Korea University, Seoul, Korea in 2005. He is now a Ph.D. student in the Department of Computer Science and Engineering at Korea University. His research interests include formal methods, model checking and process algebras.

Jungjae Lee received the B.S. degree in Computer Science and Engineering from Korea University, Seoul, Korea in 2007. Currently, he is working toward the Ph.D. degree in the Department of Computer Science and Engineering at Korea University. His research interests include formal methods, model checking and process algebras.

Jin-Young Choi received the M.S. degree from Drexel University in 1986, and the Ph.D. degree from University of Pennsylvania, in 1993. He is currently a professor of Computer Science and Engineering Department, Korea University, Seoul, Korea. His current research interests are in real-time computing, formal methods, programming languages, process algebras, security, software engineering, and protocol engineering.

Insup Lee received the B.S. degree in mathematics from the University of North Carolina, Chapel Hill, in 1977, and the Ph.D. degree in computer science from the University of Wisconsin, Madison, in 1983. He is the Cecilia Fitler Moore Professor of Computer and Information Science at the University of Pennsylvania. His research interests include real-time systems, embedded and hybrid systems, formal methods and tools, medical device systems, cyber-physical systems, and software engineering. He was Chair of IEEE Computer Society Technical Committee on Real-Time Systems (2003–2004) and an IEEE CS Distinguished Visitor Speaker (2004–2006). He has served on many program committees and chaired several international conferences and workshops, including IEEE RTSS, IEEE RTCSA, IEEE ISORC, CONCUR, ACM EMSOFT, and HCMDSS/MD PnP. He has also served on various steering and advisory committees of technical societies, including Steering Committee on ACM SIGED, CPS Week, Embedded Systems Week, and Runtime Verification. He has served on the editorial boards on the several scientific journals, including IEEE Transactions on Computers, Formal Methods in System Design, and Real-Time Systems Journal. He is a co-Editor-in-Chief of KIISE Journal of Computing Science and Engineering since Sept 2007. He is IEEE Fellow and was a member of Technical Advisory Group (TAG) of President's Council of Advisors on Science and Technology (PCAST) Networking and Information Technology (NIT). He received IEEE TC-RTS Technical Achievement Award in 2008.