A Cache-Based Model Abstraction and Runtime Verification for the Internet-of-Things Applications

Euijong Lee, Young-Duk Seo, and Young-Gab Kim, *Member, IEEE*

***Abstract*—Currently, software systems are operated in a dynamic and uncertain environment, making it difficult to predict the operating environment. Particularly, the Internet of Things (IoT) interconnects several entities, users, and information resources with services. Therefore, IoT systems can dynamically create various environments at runtime. Consequently, to support the dynamic IoT environment, an efficient verification method is required for IoT systems. Model checking is one of the formal methods for verification of concurrent systems, which is applied in several software fields. However, model checking has a chronic problem (i.e., state explosion) that obstructs the verification at runtime. To overcome this limitation, one possible approach is to abstract the system design as expression forms and then perform verification by solving the expressions. To apply this approach, the abstraction and verification processes need to be performed at a reasonable time. In this article, we focused on developing an efficient model checking method during the IoT system runtime verification. A cache mechanism is proposed that reduces the computational time for abstraction and verification, which was validated through experiments. Additionally, the comparison of other model checking tools (such as RINGA, CadenceSMV, NuSMV, and nuXmv) reveals that the proposed approach is more efficient at runtime.**

***Index Terms*—Finite-state machine (FSM), model checking, runtime, verification.**

I. Introduction

MODEL checking is an automated technique, which has a finite-state model of a system and a formal property [1], and is used as a formal method for verifying concurrent systems [2]. Therefore, model checking is applied in several software fields, such as software engineering [3], [4], robotics [5], [6], software verification [7], [8], Internet of Things (IoT) [9], [10], online software [11]–[14], and self-adaptive software [15]–[19]. Currently, software systems are operated in changeable, dynamic, and uncertain environments, and as a result, it is difficult to predict the operating environments of software systems. Therefore, verification of software needs to be performed not only at the time of design but also at runtime for checking its robustness and reliability. Particularly, IoT systems interconnect entities, users, and information resources with services. Therefore, IoT systems can create various runtime environments [20], [21]. Consequently, such systems require frequent validations and verification methods owing to variations in the runtime system configuration. Moreover, an efficient verification method is required for resource-constrained IoT devices. One of the verification methods for IoT, namely, model checking, has been applied to verify IoT's reliability during designing and runtime [10], [22]–[34]. However, model checking has a chronic problem that interrupts the verification at runtime, i.e., state explosion. The number of state space significantly increases, thus becoming a limitation for model checking [1]. For this purpose, in recent years, several studies have been conducted to overcome the limitation of runtime verification with its own characteristics (such as abstracting state machine [35], transform state machine to matrix expression [17], [18], temporal logics [5], [36], and transition model [37]). In this article, we focused on model checking using model [i.e., finite-state machine (FSM)] abstraction and model transition as equations to verify the model at runtime for the IoT applications.

To abstract models, a previous study [16] proposed a framework (i.e., RINGA) to verify software at runtime by model checking. RINGA consists of two main processes: 1) design time and 2) runtime. In design time, a system is designed as an FSM and the designed FSM is then abstracted as a simple FSM, which consists of only initial and end states. In the abstracted model, there are transitions that connect the initial and end states, and these transitions are expressed as mathematical equations. In the runtime process, the abstracted model is used for runtime verification and the verification is performed by only calculating the transitions that consist of the equations. The initial results of the runtime efficiency of the abstraction and runtime verification are presented in [16]. Additionally, RINGA has been expanded with an FSM design and game theory-based strategy extraction method for IoT [9], [10], [38]. The suitability of the expanded RINGA was demonstrated through experiments, and the results show that the framework can be applied at runtime within a reasonable computing time. Additionally, RINGA was expanded with an FSM design and a game theory-based strategy extraction method, for the IoT [9], [10]. However, RINGA has the limitation of

Manuscript received January 1, 2020; revised March 17, 2020 and May 1, 2020; accepted May 19, 2020. Date of publication May 22, 2020; date of current version September 15, 2020. This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea Government (MSIT, Development of Artificial Intelligence Based Video Security Technology and Systems for Public Infrastructure Safety) under Grant 2019-0-00231, and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) under Grant NRF-2019R1F1A1062480. *(Corresponding author: Young-Gab Kim.)*

Euijong Lee and Young-Gab Kim are with the Department of Computer and Information Security, Sejong University, Seoul 05006, South Korea (e-mail: kongjjagae@sejong.ac.kr; alwaysgabi@sejong.ac.kr).

Young-Duk Seo is with the Department of Data Science, Sejong University, Seoul 05006, South Korea (e-mail: mysid88@sejong.ac.kr).

Digital Object Identifier 10.1109/JIOT.2020.2996663

increasing the computing power in not only the abstracting model at design time but also in the verification at runtime when the model size is large and complicated. Therefore, the model checking of RINGA requires further performance improvement to handle more complex IoT systems. To overcome this limitation, we propose a cached-based abstraction and runtime verification mechanism, which reduces the overlapped calculation in the abstraction and verification processes. Experiments are conducted to compare the proposed mechanism with other model checking tools (i.e., RINGA [16], CadenceSMV [39], NuSMV [40], and nuXmv [35], [41]), and the experimental results demonstrate that the proposed method can be applied at runtime. Additionally, the cache-based approach can significantly reduce the abstraction and verification time in comparison with the initial research.

The remainder of this article is organized as follows. Section II provides a background of model checking, studies for model checking at runtime, and FSM abstraction model checking method. Section III introduces the proposed cache-based method for abstracting FSM. Section IV presents the results of the described experiments. Section V discusses the limitations and studies for model checking in the IoT systems, and Section VI presents the conclusion and future studies.

II. Background and Related Work

In this section, the background and related work are described, as well as related works on model checking at runtime. Section II-A briefly describes the model checking. Various studies that focus on model checking at runtime are described in Section II-B. In Section II-C, the FSM abstraction method, which is the initial research of this article, is introduced. In Section II-D, studies introduced that apply model checking for IoT.

A. Model Checking

Model checking is one of the verification techniques for hardware and software. Model checking uses the state-transition graph as the modeling of its target and uses temporal logic as the specification (i.e., requirement) for verification [42]. Several transition systems can be used for modeling in model checking, such as transition system [1], FSM [16], discrete-time Markov chain [17], [18], [43]–[46], and probabilistic model [47]. However, these models describe a system with states and transitions. The states denote the status of the system, and the transitions describe the connection and direction between the states. Therefore, the transition system that is used in model checking can be described as follows [1]. A transition system is a tuple (S, Act, $\rightarrow$, I, AP, L), where:

1) S is the set of states;
2) Act is the set of actions;
3) $\rightarrow \subseteq S \times \text{Act} \times S$ is the transition relation;
4) $I \subseteq S$ is the set of initial states;
5) AP is the set of atomic propositions;
6) $L : S \rightarrow 2^{\text{AP}}$ is a power of the label function.

Additionally, temporal logics are proposed to describe the specifications in model checking. In particular, linear time logic (LTL) and computation tree logic (CTL) are used to describe the specifications [2]. LTL consists of basic temporal operators: $\diamond$ (eventually), $\square$ (always), $\bigcirc$ (next), and $\cup$ (until) [1]. The temporal operators can be expressed as capital alphabets: F (eventually), G (always), N (next), and U (until) [2]. LTL is formed according to the following grammar: AP of atomic propositions (with $a \in \text{AP}$) and Boolean connectors (i.e., conjunction $\bigwedge$ and negation $\neg$) [1]

$$\delta ::= \text{true} \mid a \mid \delta_1 \bigwedge \delta_2 \mid \neg\delta \mid \bigcirc \delta \mid \delta_1 \bigcup \delta_2.$$

CTL is a branching time logic for temporal logic with basic temporal modalities [i.e., $\diamond$ (eventually), $\square$ (always), $\bigcirc$ (next), $\cup$ (until), $\exists$ (for some future), and $\forall$ (for all future)]. CTL state formulas with set AP of atomic propulsions are described as [1]

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \bigwedge \Phi_2 \mid \neg\Phi \mid \exists\delta \mid \forall\delta$$

where with $a \in \text{AP}$ and δ is a path formula. CTL path formulas are formed according to the following grammar:

$$\delta ::= \bigcirc\Phi \mid \Phi_1 \wedge \Phi_2$$

where Φ, Φ_1, and Φ_2 are state formulas. However, LTL and CTL are not comparable because their expression ranges are different. Precisely, there are properties that cannot be expressed in LTL but can be expressed in CTL, and *vice versa*. CTL can verify the existence of behavior and LTL can verify more complex behavior [1], [2]. However, there are several studies to improve the expression of model checking (e.g., CTL* [48], CTL$^+$ [49], probabilistic CTL [50], PCTL* [51], continuous stochastic logic (CSL) [52], propositional LTL (pLTL) [53], [54], quantitative temporal expressions (QuaTExin) [55], multiQuaTExin [56], signal temporal logic (STL) [5], CSL [37], and HyperLTL [57]). However, temporal logics need effective and optimized methods (algorithms) to verify that a system model can reach specific states for implementation as a software program. Therefore, in this article, we propose a cache-based abstraction method for model checking at runtime.

B. Model Checking for Runtime Verification

In this section, recent studies that focused on model checking at runtime are described. There are several domains for model checking at runtime. First, runtime model checking methods are applied at self-adaptive systems or softwares [16]–[19]. Filieri *et al.* [18] proposed a mathematical framework for self-adaptation at runtime using quantitative verification and sensitivity analysis. They focused on the probabilistic requirement, applying discrete-time Markov chains (DTMC) and quantitative probabilistic requirement description. In their framework, there are two steps: 1) precomputation (at design time) and 2) verification (at runtime). In the precomputation step, a system model, variable labels, and requirements are synthesized and translated as expressions with matrix forms. In the verification step, the expressions that represent the verification conditions for satisfaction of an adaptation system are calculated using monitored data at runtime. A wireless sensor network (WSN) system is used as a case study for the

framework, demonstrating its applicability and practical advantages. However, the framework of Filieri *et al.* has a limitation with respect to the regeneration of expressions. If a system model or the environmental factors are changed, the expressions are regenerated to adapt to the changes. Additionally, the regeneration process may require a large computational time. Nakagawa *et al.* [17] proposed a caching mechanism to reduce the number of executing Laplace expansions at runtime to overcome this problem. The caching mechanism consists of three strategies: 1) caching; 2) prediction; and 3) grouping. The experimental results demonstrated its efficacy at runtime. Additionally, Weyns and Iftikhar [19] focused on providing a guarantee for adaptation goals and proposed a modular approach for decision making in self-adaptive systems. The modular approach is based on architecture and applied stochastic timed automated (STA) as modeling. Furthermore, there is another architecture-based self-adaption study for model checking. Cámara *et al.* [58] also proposed an approach to generate optimal adaptation plans for architecture-based self-adaptation via model checking, which included stochastic multiplayer games (SMGs), a dynamic game with probabilistic transitions in game theory. Probabilistic model checking (PMC) provides modeling and analysis of stochastic behavior for SMGs in self-adaptation. Therefore, FSM with probability is used for modeling, and rPATL is applied in the approach. The approach is applied in a case study involving the infrastructure for a news website (i.e., Znn.com) [59]. Schmerl *et al.* [60] proposed an architecture-based self-protection approach with model checking based on Denial of Services (DoS). DTMC is used for formal modeling, and probabilistic reward CTL (PRCTL) is used for quantifying the results of adaptive strategies. The self-protection approach is applied to a case study that contains DoS of a website case study (i.e., Znn.con in DoS). Lee *et al.* [16] proposed a self-adaptive framework named RINGA. RINGA provides an FSM for designing self-adaptive software and a model abstraction method for runtime verification. In the abstraction method, an FSM designed for a self-adaptive system is translated as equation forms and the equations are calculated for verification at runtime. Moreover, RINGA has been applied in a case study containing IoT-based smart greenhouse scenarios [10], thereby demonstrating its extensibility.

Verification via model checking has been applied in real-time systems. Zhao and Rammig [12] focused on the state explosion problem and proposed a lightweight verification technique based on an online model checking for real-time systems to overcome the problem. They proposed a partial checking method with specific execution tracing. Therefore, their method does not directly check the execution traces but checks some steps ahead of the actual state execution. The FSM and LTL formulas are applied in the online model checking method. Qanadilo *et al.* [13] expanded the online model checking method with offline backward exploration for accelerating online model checking. The expanded online model checker has the same basic approach as that of the online model checking [12], and the Boolean satisfiability problem (SAT) solver is optimized and customized for accelerating the online model checking. Furthermore, Sudhakar *et al.* [14] researched on how to implement the online model checking method [12] into a real-time operating system and proposed an architecture for the implementation. Naskos *et al.* [11] proposed a model-driven approach for the dynamic provisioning of cloud resources, and PMC-based decision processes are used to select the optimal resources at runtime. The approach proposed a probabilistic model based on Markov decision processes (MDPs) [61], and the MDPs are monitored and they reflect the status of a target system. Moreover, the MDPs are verified online using a model checking tool (i.e., PRISM [23]), and the optimal elasticity decisions are determined at runtime.

Robotics is one of the domains that need to be verified at runtime, and studies have been conducted to apply model checking in this domain. Desai *et al.* [5] proposed a framework for building safe robots with model checking to program autonomous mobile robots with formal guarantees and high assurance. Model checking is applied for testing reactive robots that are programmed with event-driven language (i.e., P language [74]). P language comprises state machines for communication between events, and STL, which is used for the description of properties on signal values at runtime. For runtime verification, the framework for robots uses a brand of execution-driven and explicit-state model for the robotic software system. Thus, model checking may not enumerate all the states of the designed software. Additionally, Zhao *et al.* [6] developed a framework with PMC for robots deployed in extreme environments (environments that are hazardous for humans). However, PMC uses layered and parametric DTMC for system modeling and probabilistic CTL (PCTL) for describing safety and reliability properties. Bayesian estimators are used for catastrophic failure-related parameters. The PMC-based framework is used in unmanned underwater vehicles in extreme environments.

Moreover, some studies apply model checking at runtime for service-based software. Su *et al.* [37] proposed a framework called ProEva, which extends time-bounded continuous-time Markov chains (CTMCs) model checking [36] for service-based software, especially Infra as a Service (IaaS). For formalization, CSL [36] is applied in ProEva. ProEva computes asymptotic expressions and bounds for the imprecise model checkout output. A case study (i.e., Fujitsu disk drive [62]) is used for the evaluation of ProEva and the evaluation results reveal that ProEva exhibits an acceptable computational overhead. Gao *et al.* [63] also applied model checking for runtime verification of service software, especially the Web service reconfiguration architecture. In their method, the monitoring process is based on a PMC with PTCTL and linear regression analysis-based for reliability prediction. Based on the results of the monitoring process, Web services are dynamically selected. Finally, the service is reconfigured and verified by using a counterexample-guided abstraction refinement (probabilistic CEGAR) approach. Kejstová *et al.* [7] proposed an approach for the adoption of a model checker (i.e., DiVinE [64]) to perform runtime verification. DiVinE is a parallel and distributed LTL model checker for C and C++ programs. Timed automata and untimed LTL are used in the model checker. DiVinE is expanded by the addition of a run mode. In this mode, a program is explored in the standard execution order, and behavior

TABLE I
COMPARISON OF PREVIOUS RUNTIME MODEL CHECKING METHODS

Work by	Goal	Model	Temporal Logic	Target domain
Nakagawa et al. [17]	A caching mechanism to reduce computational time at runtime.	DTMC model	Probabilistic model checking	Self-adaptive System
Filieri et al. [18]	A runtime model checking with pre-computation results (symbolic expressions) in perpetual satisfaction of non-functional requirements.	DTMC model	PCTL	Self-adaptation
Weyns and Iftikhar [19]	A modular approach for decision making in self-adaptive system with distinct models and runtime simulation.	STA	Probabilistic model checking	Self-adaptive system
Lee et al. [16]	An approach for self-adaptive software with FSM modeling and verification at runtime.	FSM	Reachability of states	Self-adaptive software
Cámara et al. [58]	An approach to optimal adaptation plan generation with model checking of stochastic multiplayer games.	Finite state with probability	rPATL	Architecture based self-adaptation
Mersani et al. [60]	An approach for architecture-based self-protection with formal analysis to prevent Denial of Service (DoS)	DTMC model	PRCTL	Architecture based self-protection
Zhao and Rammig [12]	A lightweight verification technique for real-time systems with online model checking at runtime.	FSM	LTL	Dependable real-time systems
Qunadilo et al. [13]	A technique for accelerating online model checking with SAT solver as a verification engine.	FSM	LTL	Online model checking
Naskos et al. [11]	A parametric model checking method for cloud resource provisioning control at runtime.	Parametric Markov decision process	PCTL	Cloud environment
Sudhakar et al. [14]	Integration of online model checker and a small-footprint real-time operating system.	FSM	LTL	Real-time operating system
Zhao et al. [6]	A framework for probabilistic model checking on layered Markov model for verification of safety and reliability requirements	DTMC model	PCTL	Robots in extreme environments
Desai et al. [5]	A framework for building safe robots with model checking at runtime.	State machine by P	STL	Safe robotics
Su et al. [37]	A framework for runtime proactive performance evaluation called ProEva.	CTMCs	CSL	Service based software system
Gao et al. [63]	An architecture for Web service reconfiguration based on probabilistic model checking.	Probabilistic models	PCTL	Service software
Kejstová et al. [7]	A model checker for multithreaded C and C++ programs without substantial overhead into a runtime verification process.	Timed automata	Untimed LTL	C and C++ programs
Legay et al. [8]	A framework to verify bounded temporal properties for SystemC models with statistical model checking.	Probabilistic model	BLTL	SystemC
Proposed	**A method using model abstraction with a cache mechanism for verification at runtime.**	**FSM**	**Reachability**	**Self-adaptive software for IoT**

checking is performed at runtime. Ngo *et al.* [8] proposed a framework with statistical model checking for SystemC models. SystemC is a class and macro of C++ language for providing event-driven simulation interfaces. The framework monitors a set of execution traces of the model under verification (MUV), and a statistical model checker implements a hypothesis testing algorithm. The statistical model checker is implemented as a previous model checker (i.e., Plasma Lab [65]) that describes system properties as bounded LTL (BLTL).

Table I presents a summary of the comparison of previous research and the proposed method. However, each of these studies includes its distinct characteristics with various model checking methods in several target environments. In this article, we focus on enhancing the previous model checking method (i.e., RINGA [16]) for runtime verification. Therefore, we propose a caching mechanism to reduce the computing time of the abstraction and runtime processes. Details associated with the abstraction algorithm are presented in Section II-C.

C. Runtime Verification Method With Model Abstraction

In this section, the FSM abstraction method is introduced and the abstracting method is called RINGA [16]. RINGA was proposed for self-adaptive software with a modeling rule using an FSM called self-adaptive FSM (i.e., SA-FSM) and a model checking method at runtime. To apply model checking at runtime, an abstraction method was proposed in RINGA; this method abstracts a system model that is modeled as an FSM. The result of abstraction is a simple FSM called abstracted FSM (i.e., A-FSM) and the abstracted model consists of transitions described as equations. Fig. 1 illustrates the process of the abstraction method in RINGA. The abstraction process starts at the end states (e.g., *s3* and *s4* in FSM in step 1). To extract the path to the end state from the initial state, the tree data structure is used. The starting point is the end state, which is located as a root node of the tree structure. The tree structure then expands its children nodes using connected transitions in the designed FSM. If a state that matches a node in the tree structure has transitions from other states, then the transitions are used for the expansion of the tree structure. For example, in Fig. 1, *s3* is set as a node of the tree structure and has transitions (*e2* and *e5*) from other states (*s1* and *s2*); thus, *s1* and *s2* are set as the children nodes of *s3* in the tree structure. However, if there are iterative states, the iterations are deleted and the expansion is stopped. Finally, the tree structure has root nodes from the end states in the system model, and the terminal nodes are the initial states. Furthermore, the paths from the terminal nodes to the toe root node in the tree structure denote the paths from initial states to the end states in the

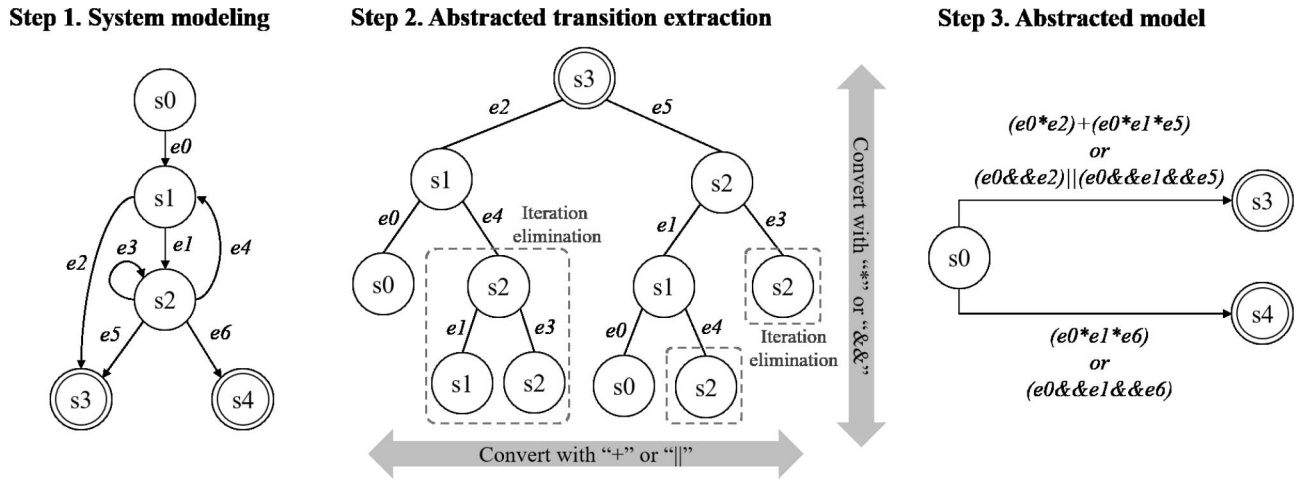


Fig. 1. FSM abstraction process in RINGA [16].

system model. The results of the tree structure are translated as equation forms.

In the translation process, the parent relationships are converted into * or && operations because the parent relationship indicates the components of a path. If one edge (transition) cannot reach the next state, the remained states (connected states) also cannot be reached. Additionally, the sibling relationship is translated as + or ‖ operations because the sibling relations imply different paths. If one sibling is unable to reach an end state, but the other sibling may be able to reach the end state. If the transitions of SA-FSM are described as Boolean type, && and ‖ are applied. Furthermore, if the translations are expressed between 0 and 1 (i.e., probability), + and * operators are applied. After the translation process, the abstracted translations are applied as translations of A-FSM. The extracted A-FSM is used at runtime for verification. The effectiveness of RINGA for verification at runtime is demonstrated through empirical evaluation and case study. The details of the FSM abstraction are described in the previous research, along with the algorithms and definitions [16].

However, RINGA has a limitation when the computing power needed for the abstraction algorithms and model size is increased. Additionally, even if the abstracting process eliminates duplicated calculation by deleting the iterative paths, there may exist duplicated calculation in verification at runtime. To overcome these limitations, a cache-based FSM abstraction method and verification are proposed in this article.

D. Model Checking in the IoT Systems

Several studies employed different model checking tools for modeling and verification of the IoT systems. A model checking-based self-adaptive framework was proposed for the IoT applications [9], [10]. The self-adaptive framework was based on the RINGA model checker. The framework proposed an FSM-based IoT system design and verified its requirements following the designed FSM reachability. A model checking applied IoT system was proposed for microgrids [22], wherein a localized group of electricity sources was centered on distributed power sources from the traditional wide-area synchronous grid (i.e., a macrogrid). The IoT system consisted of four layers: 1) sensors; 2) communication devices; 3) controllers; and 4) actuation. Each layer contained probabilistic models that use model checking tools, such as a probabilistic model checker (PRISM) [23], for verification. The possibility model checking-based approach was demonstrated to improve reliability in the microgrid IoT system. A probabilistic position estimation and model checking approach was introduced for resource-constrained IoT devices [66]. The approach estimated the IoT end node positions using a Markov localization algorithm. Furthermore, the model checking technique was applied for position validation. The process meta language (PROMELA) and SPIN model checking tool [67] were employed as modeling language and verification tools, respectively. A positive indication was observed for the proposed estimation approach in a cattle-breeding IoT network. A reference model was proposed for IoT application-specific design and analysis challenges, such as representing heterogeneous layers (hardware and software components) and lack of Quality-of-Service (QoS) verifying methodologies [24]. The reference model was named SysML4IoT and based on the ISO/IEC/IEEE 15288 [68] IoT reference model, and a translator was proposed to verify the model with a model checking tool. To apply the modeling results into the model checking tools, such as a new symbolic model checker (NuSMV) [40], [69], a model-to-text translator (SysML2NuSMV) was used to translate SysML4IoT into a text. Furthermore, the approach was applied to an IoT scenario for building energy conservation to evaluate the proof of concept. Furthermore, two IoT service-oriented computing (SOC) challenges: 1) unreliable service and 2) resource constraints, were resolved using PMC, which consisted of modeling and analysis [27]. An IoT service composition was functionally modeled as FSM and was translated to the MDP to specify the service operator reliability. Furthermore, the analysis included PCTL to specify service composition quality and PRISM to check probabilistic modeling. This approach was applied to a simple fire alarm service in a meeting room. A Petri net in a Petri net PN^2 system was proposed for the design and analysis of the IoT-based multiagent services [28]. Moreover, the

multiagent system was analyzed by the SPIN model checking tool. The multiagent system was applied to an IoT service in a smart home case study. Furthermore, the NuSMV model checking tool was applied to support PN^2 in a conference support system case study [29]. A hybrid testing environment was proposed for IoT systems that have various system configurations and nondeterministic execution [25]. The model checking method was used for the automatic verification of communication-derived possible execution orders. The hybrid testing environment applied the SPIN model checker and evaluated it in the regression test of a heating, ventilating, and air-conditioning (HVAC) IoT system. The regression test exhibited an effective reduction in the IoT system regression test burden. A formal colored Petri nets (CPN) model [32] of the message queuing telemetry transport (MQTT) protocol [70] was proposed for the IoT network. The CPN model applied incremental model checking that exhibited reduced state explosion effect than the previous model [71].

Moreover, the model checking method was applied for security-related issues, such as uncertainty, preventing undesirable interaction, risk analysis, security framework, and functional correction. The ThingML framework [72], [73] provided modeling and language for heterogeneous and distributed systems. Furthermore, ThingML could assist the rapid development of resource-constrained IoT applications. However, ThingML exhibited model capability-related limitations, such as uncertainty and quantification of the models. A ThingML-based IoT framework was proposed to quantitate the evaluation uncertainty [31]. The ThingML-based framework received the design requirements, which contained user-expected performance matrices, and extended ThingML designs, along with various information maps, such as network delay and sensor inputs. The framework translated the design requirements as temporal logic-based queries. As the network of priced timed automata (NPTA) models were the computational models for ThingML language, the extended ThingML designs were transformed into NPTA models. Furthermore, NPTA models evaluated the QoS with model checkers such as UPPAAL-SMC [74]. To examine the effectivity of the framework in QoS evaluation, the framework was applied to two cases: 1) interaction of things within client/server architecture and 2) a heating control system. A framework named IOTSAN was proposed, using model checking, to prevent undesirable interactions that result in unsafe physical states in IoT systems [30]. Moreover, IOTSAN was designed to prevent the state explosion. The application source code in IOTSAN was translated to PROMELA form, to facilitate the model checking tools, such as the SPIN. An attribution mechanism identified problematic and potentially malicious applications. Test cases were used to assess the efficiency and identify a safe property violation. Furthermore, a framework named IoTRiskAnalyzer framework, which employed PMC was proposed to analyze IoT system configuration risks [26]. This framework used vulnerability scores from risk assessment models, candidate IoT configurations for achieving a system's goal, and capabilities as input, and produced an ordered set of risk exposure probabilities-based configurations. The PRISM probabilistic model checker was used for the analysis. IoTRiskAnalyzer efficiently analyzed the risk for IoT network, in a home security automation scenario. Keerthi *et al.* [33] applied to model checking to solve security-related issues, such as functional correctness of implementation, programming bugs, side-channel analysis, and hardware Trojans. A C-based model checker (CBMC) [75] was implemented in the security issues. A formal verification demonstrated that CBMC was effective in some issues, such as programming bug detection and side-channel security implementation. However, formal verification tools were required to resolve the huge state space. Furthermore, a security framework was proposed for the IoT system modeling and analysis [34]. In the security framework, formalism described IoT structures and components, including entity interactions and activities. The formalism was transported into PRISM language to check requirements that describe PCTL. Moreover, the framework applied a smart healthcare emergency room to exhibit security framework's effectiveness.

Table II summarizes the existing IoT model checking methods. Several IoT studies applied model checking methods with different efficiencies. However, we focused on developing an efficient model checking method during the IoT system runtime verification. We have aimed at improving the abstraction and runtime verification method used in the RINGA model checking tool. RINGA has been applied in the IoT modeling and runtime verification [9], [10]. However, this required greater computing power for larger models and was complicated. Therefore, RINGA exhibited limitations in complex IoT environments. Consequently, efficient abstraction and runtime verifications were needed for divergent IoT environments. Therefore, we proposed a cache-based model abstraction and verification method.

III. Caching Mechanism for Enhancement of Model Abstraction and Verification at Runtime

The initial result on runtime verification with model abstraction is called RINGA [15], [16]. This initial research provides an FSM design to model self-adaptive software and runtime verification with model abstraction. Moreover, the research is applied to the IoT environment [9], [10]. However, RINGA required performance improvement to be applied with complex FSM especially in the design-time process (i.e., the model abstraction process). Therefore, to solve the limitation of RINGA, we proposed a caching mechanism to enhance the performance of not only model abstraction but also runtime verification compared to the previous model checking method.

A. Overview

The proposed cache-based model checking method consists of two phases: 1) abstraction at design time and 2) verification at runtime. Fig. 2 shows an overview of the proposed approach.

The abstraction phase is responsible for model abstraction and the result of model abstraction is used in the verification phase. The abstraction phase can be divided into two parts: 1) system modeling and 2) model abstracting. In system modeling, a system that needs runtime verification is designed as an FSM. Note that the abstracting phase only

TABLE II
EXISTING IOT MODEL CHECKING METHODS IN IOT

Worked by	Goal of the Study	Models Employed	Model Checking Tools Employed
Lee et al. [9], [10]	A self-adaptive framework for designing and verification, using model checking for IoT.	FSM	RINGA
Liang et al. [22]	A model checking-based microgrid IoT framework for reliability enhancement.	Probabilistic model	PRISM
Sekizawa et al. [66]	Probabilistic position estimation and model checking for resource constrained IoT devices.	PROMELA	SPIN
Costa et al. [24]	A framework with reference model and model checking-based QoS verification for IoT application design and analysis.	FSM	NuSMV
Kuroiwa et al. [25]	A hybrid testing environment with execution test and the IoT model checking.	PROMELA	SPIN
Mohsin et al. [26]	A framework for the IoT system risk analytics, using probabilistic model checking.	Probabilistic model	PRISM
Li et al. [27]	Modeling and analysis of the reliability and cost of service composition in the IoT system, using probabilistic model checking.	FSM, probabilistic model	PRISM
Yamoguchi et al. [28]	A system to design and analyze the IoT-based multi-agent service, with model checking.	PROMELA	SPIN
Nakahori and Yamaguchi [29]	Same as Yamoguchi et al. [28]	FSM	NuSM
Nguyen et al. [30]	A framework to prevent unsafe physical status of the IoT systems.	PROMELA	SPIN
Xu et al. [31]	A framework to quantitate uncertainty evaluation for ThingML-based IoT design, using model checking.	Timed automata	UPPAAL-SMC
Rodriguez et al. [32]	A formal model for the IoT protocol, using incremental model checking.	-	-
Keerthi et al. [33]	Formal verification for security-related issues.	-	CBMC
Ouchani [34]	A security framework for the IoT modeling and analysis.	Probabilistic model	PRISM
Proposed	**A cache-based model abstraction and checking method to verify the IoT applications efficiently**	**FSM**	**RINGA**

considers how to abstract the designed FSM during system modeling. Therefore, the details of the system model design are out of the scope of this article. However, a simple FSM design is proposed for a caching mechanism called cached FSM (C-FSM) and the proposed FSM contains rules for the abstracting phase. The proposed caching model is applied to the system model in the abstracting phase. In the abstraction process, the designed system model is abstracted as a model for verification at runtime, and the abstracted model is called cached and abstracted FSM (CA-FSM). Note that RINGA's approach is used when translating from a path to an abstracted transition (i.e., equations form of the path) (see Section II-B). Furthermore, transitions of the system model are parameterized and saved in the cache database. Extracted CA-FSM transitions are also saved in the cache database. Finally, the abstracted model is transferred to the verification phase and used for verification at runtime with the cached database. Details of the abstraction phase are described in Section III-B.

The verification phase is executed at runtime. To verify the system at runtime, the proposed approach is made with a loop. The loop is based on a MAPE loop mechanism (monitoring, analysis, planning, and execution) that is a prominent control loop to organize self-adaptive software [76], [77]. However, the planning and execution processes are combined as one process in the proposed approach. The proposed loop is made up of three processes: 1) monitoring; 2) verification; and 3) adaptation. The monitoring process is responsible for collecting data from the system and the environment and monitored results are used to update parameter values in the cached database. The verification process is responsible for the verification of the system. The updated parameters in the monitoring process are used to update cached translations, and the translations are utilized for verification. According to the verification result, the condition of the system is analyzed and appropriate strategies are triggered in the adaptation process. Subsequently, the monitoring process is executed, and the loop continued after the adaption process. Details of the verification phase are presented in Section III-B.

B. Caching Mechanism

In this section, the caching mechanism is described. As described in the previous section, the caching is performed at design and runtime phases. The caching mechanism in the design-time phase is described in Section III-B1 and the caching mechanism in the runtime phase is described in Section III-B2.

1) Caching in Design Time: The proposed approach verifies a system that described an FSM at runtime, thus the system must be described as an FSM. Additionally, the change of transitions from the FSM must be monitored to verify the system at runtime. Therefore, all transitions are extracted as parameters and the parameterized transitions are saved into a parameter database. Values of the parameters (i.e., change of the transitions) are monitored at the runtime. Fig. 3 shows the parameterization process.

After the parameterization, the designed system model is abstracted for verification at runtime. The abstraction is based on RINGA's approach (see Section II-B), but the proposed approach applies the cache mechanism for performance improvement. However, to apply the cache mechanism, the system model is being remodeled as C-FSM. Note that the

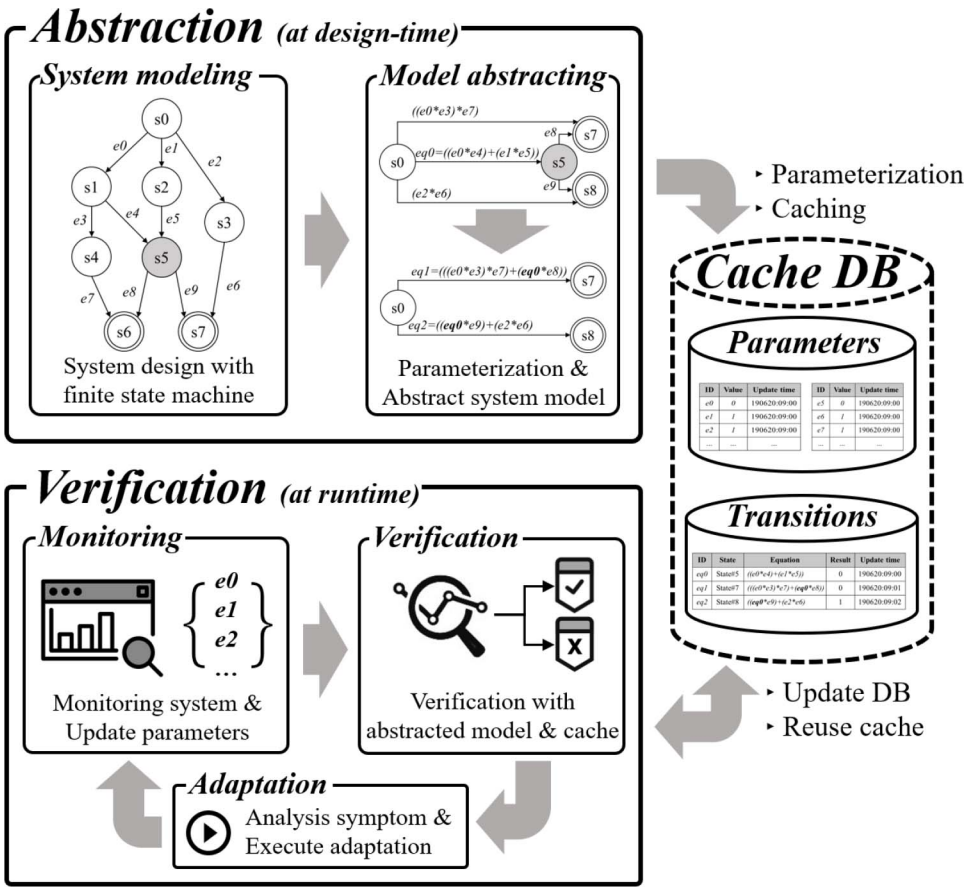
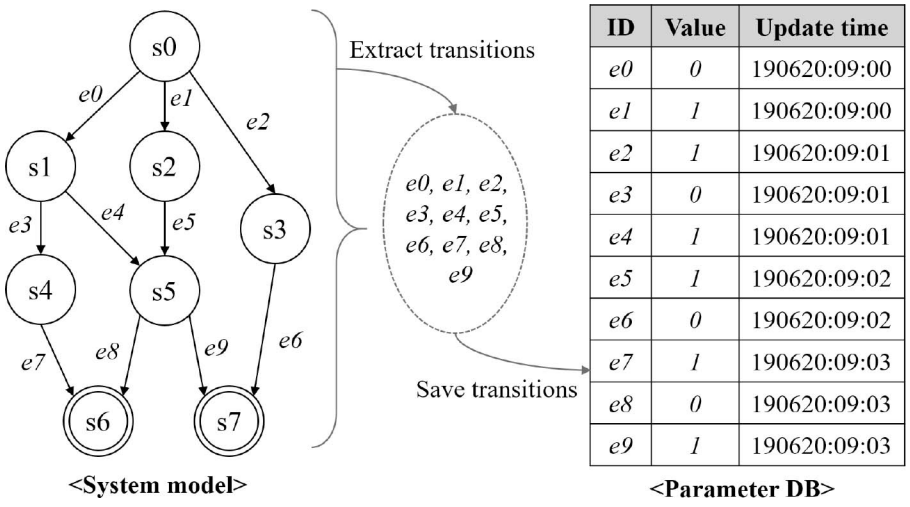


Fig. 2. Overview of the proposed approach.

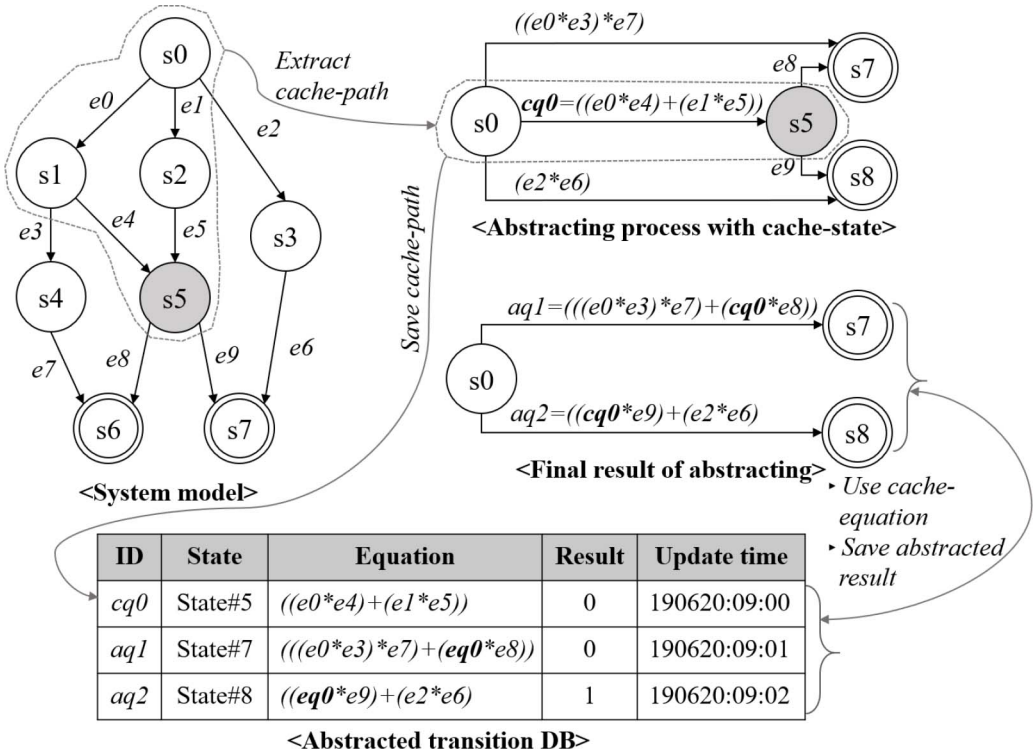


ID	Value	Update time
e0	*0*	190620:09:00
e1	*1*	190620:09:00
e2	*1*	190620:09:01
e3	*0*	190620:09:01
e4	*1*	190620:09:01
e5	*1*	190620:09:02
e6	*0*	190620:09:02
e7	*1*	190620:09:03
e8	*0*	190620:09:03
e9	*1*	190620:09:03

Fig. 3. Transition parameterization.

remodeling is not impacted at the system model but is only for applying the cache mechanism. We defined types of transitions as in-transition and out-transition. In-transition is a transition that comes into a state from another state (i.e., if there is a state, and there is a transition from other states into the state, then the transition is in-transition). On the other hand, out-transition is a transition that goes ahead to other states (i.e., if there is a state, and there is a transition from the state to other states, then the transition is out-transition). With this definition of transitions, the C-FSM can be expressed as a tuple (S, s_0, $\rightarrow$, I, AP, L), where:

1) S is the set of states;
2) the states classified into three types $\{S_{\text{normal}}, S_{\text{cache}}, S_{\text{end}}\} \subseteq S$;
3) s_0 is an initial state;
4) S_{end} is a set of end states;



ID	State	Equation	Result	Update time
cq0	State#5	*((e0*e4)+(e1*e5))*	0	190620:09:00
aq1	State#7	*(((e0*e3)*e7)+(**eq0***e8))*	0	190620:09:01
aq2	State#8	*((**eq0***e9)+(e2*e6)*	1	190620:09:02

<Abstracted transition DB>

Fig. 4. Caching in the abstracting phase.

5) S_{cache} is a set of states which have two or more out-transitions;
6) S_{normal} is a set of states which is not an initial state, S_{cache} and S_{end};
7) $\rightarrow \subseteq S \times S$ is the transition relation;
8) AP is the set of atomic propositions;
9) $L : S \rightarrow 2^{\text{AP}}$ is a power of the label function.

C-FSM has a distinctive state type that is S_{cache}, and the state set is used as criteria for which transition can be cached. Fig. 4 shows how to use S_{cache} for caching mechanism in the abstraction process. There are two end states *s6* and *s7* and a cache state *s5*. The path to reach the cache state is needed to extract a path to reach each end state (see "abstracting process with cache state" in Fig. 3). Therefore, the cache path is required to extract both end states. The path *cq0* is thus saved in the cache database to prevent repetitive extraction of the cache path. In other words, if an abstraction is to find a path to reach state *s7*, then save a path to reach *s5*. The cached path is then used as an adaptation process to find a path reaching *s8*. However, the other paths that are not cached path are not being cached, because the other paths are not needed for repetitive calculation (i.e., the other paths are only calculated once in the abstraction process). Finally, the abstracted results that contain paths reaching end states are saved in the cache database.

The abstracting result can be described as a simple FSM, and we named the FSM as CA-FSM. CA-FSM is described as a tuple (S, s_0, $\rightarrow$, I, AP, L), where:

1) S is the set of states;
2) s_0 is an initial state;
3) all states are end states except initial state;
4) $\rightarrow \subseteq S \times S$, and it is only one type $s_0 \times S$;
5) all transition is described as an equation expression;
6) AP is the set of atomic propositions;
7) $L : S \rightarrow 2^{\text{AP}}$ is a power of the label function.

As described in the definition, there is only one transition relationship in CA-FSM and the relationship is described as an equation expression. Moreover, the equation expression contains paths to reach the end states of CA-FSM.

Algorithm 1 lists the pseudocode of the abstraction phase. Input data are an FSM describing a system that needs verification at runtime. Output data are a CA-FSM, and the output data are used for verification at the runtime phase. The algorithm consists of two loops (i.e., lines 2–4 and lines 5–7 in Algorithm 1). The former loop repeats for the number of transitions in the input FSM. In the loop, all transitions are parameterized and saved in the cache database (see Fig. 3). In the second loop, the loop repeats for each end state to extract the abstracted path reaching each end state. To find the paths, a function is called to extract paths, and Algorithm 2 shows the pseudocode of the path extraction (i.e., "searchNext" at line 6 in Algorithm 1). Input data are FSM that needs to be abstracted and state id that needs to be searched currently. There is a loop that repeats for the number of out-transitions of the inputted state (i.e., "stateNow" in Algorithm 2). In the loop, if an out-transition is connected with the initial state, then the transition is saved as a path (lines 3 and 4 in Algorithm 2). Furthermore, there are two options when the connected state with the transition is the normal state (lines 6–12 in Algorithm 2). If the connected state is already existing in the cache database, then the cached path is added in the return path (lines 7 and 8 in Algorithm 2). Otherwise, if the connected state is not a

Algorithm 1 Pseudocode of the Abstracting Phase With Cache

Input: System model as finite state machine(FSM)
Output: Abstracted finite state machine(CA-FSM)
1: $CA - FSM = null$;
2: **for** FSM.nextTransition **do**
3: $cachedDB.papameter(transition)$;
4: **end for**
5: **for** FSM.hasEndState **do**
6: $CA - FSM+ = searchNext(FSM, endStateNow)$;
7: **end for**
8: **return** $CA - FSM$

Algorithm 2 Pseudocode of Path Extraction

Input: System model as finite state machine (FSM), current state id (stateNow)
Output: Path consisted with transitions
1: $returnPath = null$;
2: **for** stateNow.hasInTranstion **do**
3: **if** $stateNext = initialState$ **then**
4: $returnPath+ = stateNow.outTranstion$
5: **else**
6: **if** $nextState = normalState$ **then**
7: **if** $cacheDB.has(nextState)$ **then**
8: $returnPath+ = cached.path(nextState)$;
9: **else**
10: $returnPath = stateNow.outTranstion + searchNext(nextState, FSM)$;
11: **end if**
12: **end if**
13: **end if**
14: **end for**
15: **if** stateNow = cacheState **then**
16: $cached.add(stateNow, path)$;
17: $returnPath = cached.path(stateNow)$;
18: **end if**
19: **return** $returnPath$;

cached state, then the algorithm calls itself recursively with the connected state as input data (lines 9–11 in Algorithm 2). The result of the recursively called function is the path from the initial state to the input state, thus the returned path is saved with out-transition that currently used transition in the loop (line 10 in Algorithm 2). After the loop, if the inputted state is a cache state, the extracted path is saved in the cache database and the returned value is updated as a unique id of the cached database. The result is returned to Algorithm 1 (line 19 in Algorithm 2). Finally, after the loop with end states is ended, the abstracted FSM is returned (line 8 in Algorithm 1).

2) Caching in Runtime: An abstracted FSM (i.e., CA-FSM), cached parameters, and cached transitions are used in the runtime phase. The phase is made up of the same components as a loop with three processes: 1) monitoring with parameter updating; 2) verification; and 3) adaptation. Fig. 5 shows the processes in runtime verification. In the monitoring process, parameters that were extracted at the design-time phase are updated and the value of parameters is updated in the cache database. The updated parameters are used in the verification process when calculating transitions of CA-FSM. Additionally, the verification process uses a cache database for model checking with CA-FSM. As described in the previous section, CA-FSM contains reachable paths to reach the end states of the designed system model and results of CA-FSM's transitions contain model checking results. However, the first step of the verification process is calculating the cache equation and updating the result of the calculation in the cache database. This is because the cache equation is used at least twice for verification (i.e., abstracted transitions will use cache equations at least twice). After updating cache equations, abstracted transitions are calculated and the updated cache equations are used if the calculation needs the cache equations. For example, in Fig. 5, *cq0* (cache equations) is calculated and updated first, then *aq1* and *aq2* (abstracted transitions) are calculated with the result of *cq0*. After the verification process, verification results are transferred into the adaptation process, and the adaptation process performs analysis symptoms using the results from the verification process. Finally, the monitoring process is executed and the loop of runtime phase is continued.

Algorithm 3 shows the pseudocode of the runtime processes. Input data are CA-FSM, cached parameter list *parameters*, cached equations *cacheEqus*, and abstracted transitions *absTrans*. The algorithm is made up of an infinite loop (lines 2–15), but it can be terminated if a changing model is needed or runtime errors occur. In this article, we focused on the cache mechanism for verification at runtime, thus those details of termination conditions are out of scope. However, parameters are monitored and the monitored results are saved in the cache database (i.e., values of parameters are updated in the cache database) (lines 2–5). After updating the parameters, the cache equations are updated to prepare the calculation of the abstracted translation using the updated parameter values (lines 5–8). After updating cache equations, the abstracted transitions are calculated using updated parameters and cache equations, and the results are updated in the cache database (lines 9–12). Finally, the results of abstracted FSM (i.e., results of model checking) are used for analyzing a system, and if adaptations are needed, triggers are selected and executed (lines 13 and 14). After the executions, the loops are continued.

IV. Empirical Evaluation

This section discussed a set of experiments for the empirical evaluation of the proposed caching mechanism. A prototype of the proposed approach is implemented using Java 12.0.1. The experiments are conducted on Windows 10 desktop equipped with an Intel Core i5-9400F (2.90 GHz with six cores) and 16-GB memory. The proposed approach consisted of phases, which are design time and runtime, and the design-time phase is executed only once. Therefore, we divided the evaluation into design-time and runtime performance. The proposed approach is based on a previous model-checking tool (i.e., RINGA [15], [16]), which involves the abstracting process to reduce the runtime verification time. Therefore, the abstraction performance of the proposed approach is

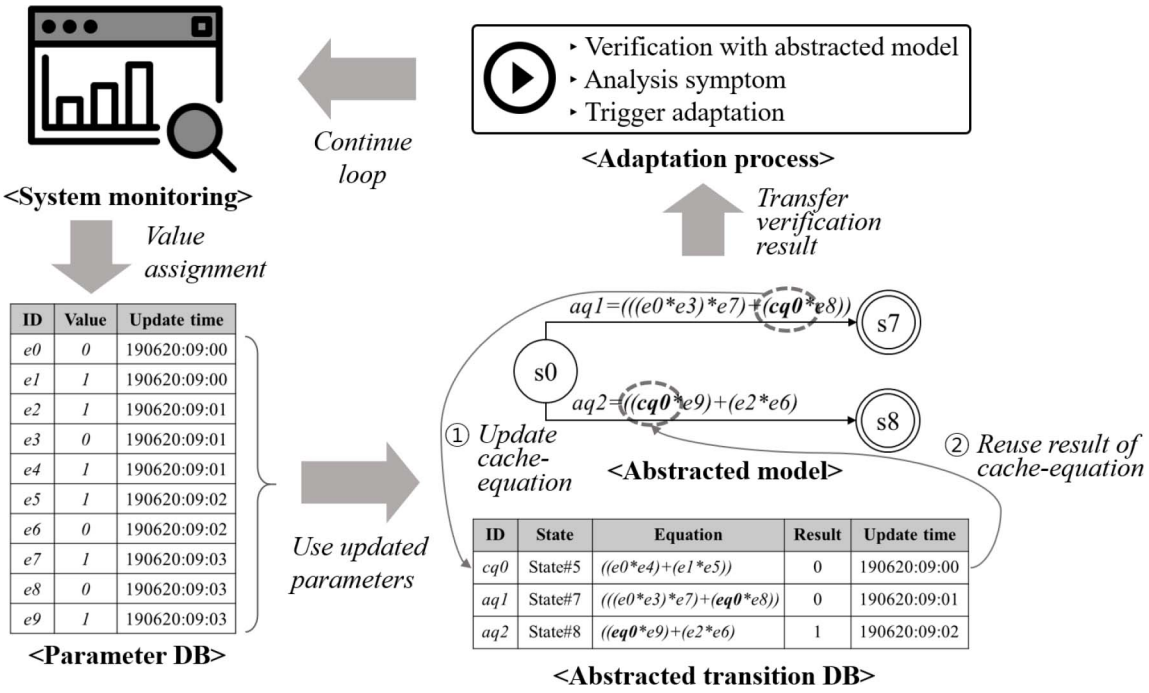


Fig. 5. Caching in the runtime phase.

Algorithm 3 Pseudocode of the Abstracting Phase With Cache

Input: Abstracted finite state machine (CA-FSM), cached parameter list (parameters), cached equations (cacheEqus), abstracted transitions (absTrans)

Output: None

```
for true do
    for parameters.hasNext do
        updateDB(monitor(parameterNow));
    end for
    for cacheEqus.hasNext do
        resultEqu = calculate(equNow, parameters);
        updateDB(equNow, resultEqu);
    end for
    for absTrans.hasNext do
        resultTran = calcualte(absTranNow,
        parameters, cacheEqus);
        updatdDB(absTrannow, resultTran);
    end for
    triggers = analysis(cacheDB.getAbsTrans);
    execute(triggers);
end for
```

compared with the previous research in design-time evaluation. Moreover, in the runtime performance evaluation, the proposed approach is compared with the model checking tools (i.e., Cadence SMV [39], NuSMV [40], [69], nuXmv [35], [41], and RINGA [15], [16]). The details are provided in the following sections.

A. Design-Time Performance

To perform the experiment, we randomly select well-formed FSMs of two different types to evaluate design-time performance. The first experiment data set is generated with increasing states, fixed transitions, and three end states. Each state in the FSMs has two out-transitions that denote connection into other states. The out-transitions are randomly assigned to states; however, every state has at least one in-transition to maintain the connection of the FSMs. Finally, every state is connected and must be explored to abstract the FSMs. The number of states is increased from 5 to 50, and 100 FSMs randomly generated at each state number. Fig. 6 shows results obtained for increasing number of states. Above all, time for model abstraction is significantly reduced after 30 states. The result shows that the proposed caching approach can effectively reduce the computational time for model abstraction compared to RINGA. Naturally, more computational time is required when the number of states is increased, but the proposed approach spends reasonable time at runtime (i.e., only 1.23 ms with 50 states).

Furthermore, we performed a similar experiment, and the data set of the second experiment is made up of increasing transitions and 15 states. The former experiment data set reflects the change of model size, this experiment data set reflects the change of model complexity. Each of the 15 states is assigned various number of transitions from 2 to 5, one transition is used to connect every state (i.e., every state has at least one in-transition), and the remaining transitions are randomly assigned. The results with the increasing number of transitions are shown in Fig. 7. Similar to the previous result, more computational power is required when model complexity is increased. However, these results also denote that the proposed caching approach effectively reduces computational time for model abstraction compared to RINGA (i.e., only 0.2 ms with five transitions). Therefore, the proposed approach can handle complex model design compared to RINGA.

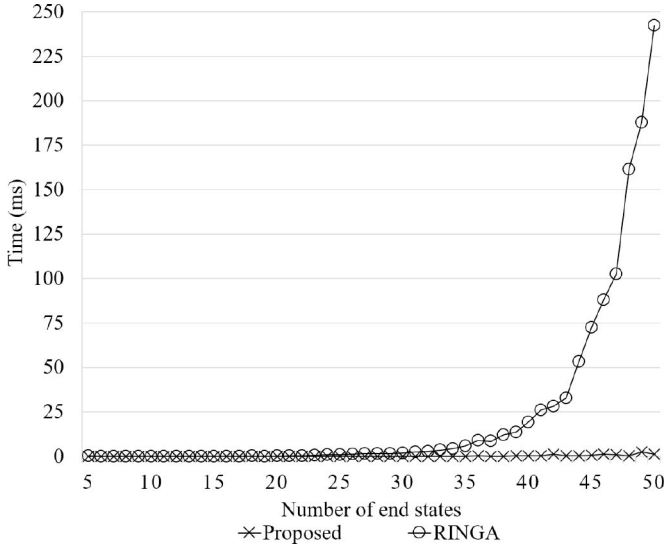


Fig. 6. Results of model abstraction with increasing states (with two transitions and three end states).

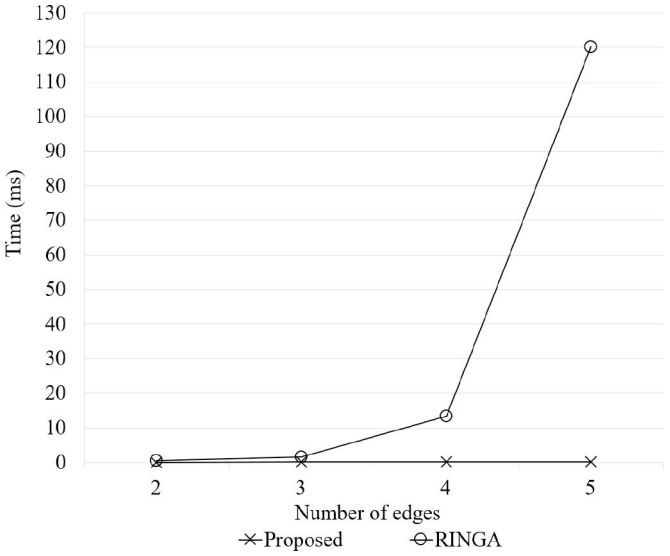


Fig. 7. Result of model abstraction with increasing transitions (with 15 states and three end states).

As described before, the proposed approach is based on RINGA, and the previous research has a specific limitation about computing power for abstracting large-scale models. Notwithstanding the model abstraction process is performed only once in design time, this limitation makes it difficult to apply in low computing devices (e.g., mobile or IoT device). Additionally, if a system model is changed frequently, the limitation can occur thus wasting computing power for the abstracting process. However, the results of the design-time evaluations suggest that abstraction ability and stability are significantly improved compared to RINGA; therefore, the proposed approach clearly resolves the limitation of RINGA.

B. Runtime Performance

In this section, experiments were performed to evaluate runtime performance. The proposed approach is compared with

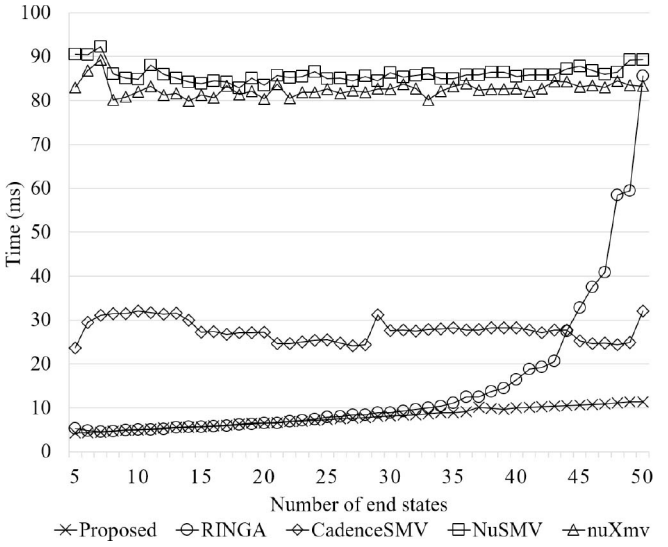


Fig. 8. Results of runtime performance with increasing states (with two transitions and three end states).

previous model checking tools (i.e., RINGA, Cadence SMV, NuSMV, and nuXmv). To briefly explain the model checking tools, RINGA is a base model of the proposed approach without a caching approach. Cadence SMV is a symbolic model checker tool that was released by Cadence Berkeley Labs, and NuSMV is an open-source version of cadence SMV [42]. Moreover, nuXmv is also a symbolic model checking tool for analysis of synchronous finite- and infinite-state systems [35], [41]. These model checkers are powerful tools in the model verification area. However, those data sets are used for design-time experiments (see Section IV-A). The caching approach and RINGA use abstracted results from the design-time phase because abstracted models are used at runtime in both model checkers. The other tools use the same FSM model with reachability from the initial state to end states. Using the intuitive semantics of temporal modalities [1], reachability with n end states is described as follows:

$$\delta = \bigwedge_{i=1}^{n} \exists \Diamond endState_i.$$

Symbol "$\exists$" denotes "exists," "$\Diamond$" indicates "eventually," and "$\wedge$" indicates conjunction; therefore, "$\exists \Diamond endState$" will be "true" if there is a path from initial state to endState. Finally, δ is true if there are paths to reach all n states. However, FSMs in the test set are generated without nonconnected states, thus there are always paths to reach all end states. Therefore, δ is always true in the data set, thus reachability to each end states must be checked (i.e., reachability of every end states should be performed).

Runtime performance results are shown in Figs. 8 and 9. As shown in the results, the proposed approach significantly improves runtime performance compared with RINGA. Especially, in the case of RINGA, the model increases rapidly when it becomes complicated or large while the proposed method is stable even if the model change is complex. However, the results show that more computational time

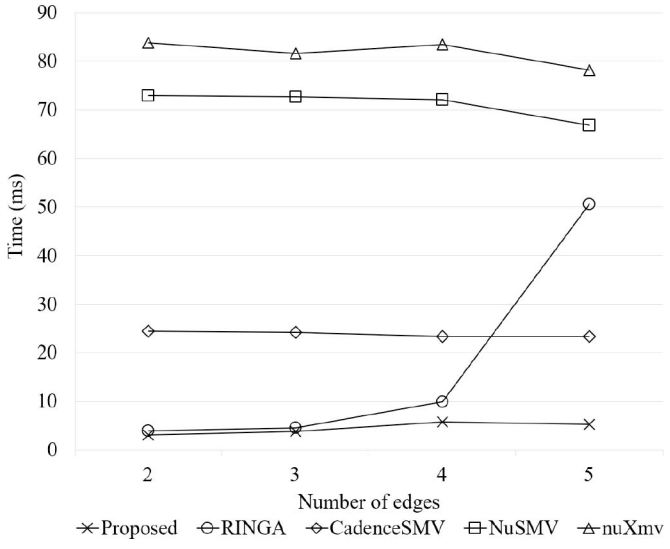


Fig. 9. Result of runtime performance with increasing transitions (with 15 states and three end states).

is required when the model size and complexity increased because the abstraction equation (i.e., abstracted equation) is more complicated as the size and complexity increase. On the other hand, other tools show monotonic computational time even if model size and complexity are increased. The reason is that Cadence SVM, NuSMV, and nuXmv use different methods to find reachability. The tools terminate model checking if they find a path to reach end states (i.e., the tools only provide a single path to reach end states). However, the proposed approach considers all paths to reach the end states because the abstracted equations contain all possible paths to reach the end states. Nevertheless, the caching approach is better than the path finding of other model checking tools.

V. Discussion

In this article, we proposed the cache-based model checking method with model abstraction and runtime verification for IoT applications. The proposed approach yielded excellent experimental results. However, there are several problems to be solved before moving forward. In this section, we present a discussion on the limitations of the proposed approach and future research to overcome the limitations.

The proposed approach only considers the reachability of FSM to verify an IoT system at runtime; thus, the requirements of the system must be designed with consideration of reachability. This limitation causes constraints of expression to describe requirements and restrictions design of FSM. To overcome this limitation, temporal logics (i.e., LTL and CTL) can be applied in the proposed approach. Temporal logics can be combined with several patterns; therefore, temporal logics are a method that needs to be applied in selecting cache states for optimal abstraction. Additionally, if temporal logics are applied in the proposed approach, the FSM model can enhance the design of IoT environments in comparison with the previous research [10]. However, if this limitation is overcome, we can expect some advantages. IoT modeling is less restricted than in the previous research; however, richer requirement expression is possible.

Here, we focused on improving the model checking for the IoT runtime verification. However, the proposed approach required referenceable IoT system models. Previous investigations [10], [11] provided a framework and reference design of FSM for our initial investigation such as applying the RINGA algorithm. However, as the abstraction and verification methods were different, the previous framework and design were inappropriate for the cache-based model checking approach. Therefore, novel reference model designs and frameworks were needed to apply the cache-based models in the general IoT environments.

IoT connects several objects (such as device, user, and requirements), and IoT environments can be changed dynamically at runtime [78], [79]. Therefore, system design may be changed at runtime, and the change must be reflected in the system for proper verification. In the proposed method, the system is designed as an FSM and the designed FSM is abstracted as a model for runtime verification in design time. Therefore, if the system change is needed, the whole abstraction process is performed to reflect the change. This can cause unnecessary computing power in the abstraction process because the system change may not need to revise the overall system model. Therefore, the proposed approach needs a method to modify a part of the designed model for reflecting the changes at runtime. However, we assumed that system model updating is possible only in some cache states changing without the change of the overall model. In other words, the change of the IoT system may be reflected by the revisioning of some part of the overall model; thus, some cache states and transitions that are related to the change need to be updated. Additionally, it is assumed that if partial revision is possible, the computing power can be decreased than the abstracting overall FSM.

Moreover, as the proposed approach is aimed at IoT systems, it requires additional improvements before its application in real IoT systems. We referred to studies that were conducted to improve methodologies for implementation [80]–[82]. To improve our approach, we plan to develop a process that comprises three phases: 1) analysis; 2) design; and 3) implementation. In the first phase, we will perform analysis to enhance the performance of the verification method by applying temporal logics and abstraction at runtime, as described in the previous paragraphs. In the design phase, we will develop a high-level system design based on FSM and the proposed verification method. Furthermore, the system will be designed using self-adaptive concepts and mechanisms (e.g., MAPE loop) such that it can be adapted to dynamic IoT environments at runtime. In the implementation phase, we will conduct a simulation and implementation with a real-IoT system. The simulation will overcome the limitation of the system design and verification performance in different IoT environments (e.g., the complexity of the IoT environment with increasing actuators or sensors). The implementation will be conducted with concrete IoT use cases for smart greenhouse scenarios.

VI. Conclusion

In this article, we proposed a cache-based FSM abstraction and runtime verification method for IoT and solved problems of the initial research [15], [16]. The problems increased the computing power with complicated FSMs; thus, the initial research is difficult to apply in complex IoT systems. The cache mechanism consisted of two phases: 1) abstraction and 2) verification. The system model is provided as an FSM, and the abstraction phase parameterizes and abstracts the system model. In abstraction, the system model (i.e., C-FSM) is abstracted as a simplified FSM (i.e., CA-FSM), and the abstracted transitions are extracted. The results of the parameterization and abstraction are saved in a cache database. Furthermore, in the verification phase, the system is first monitored and the parameters are updated in the database. The abstracted transitions are updated and used for runtime verification, and adaptation is performed based on the results of the verification. The proposed approach is implemented and compared with not only the initial research but also other model checking tools. The experimental results demonstrate that the approach significantly reduces the computational time for abstracting FSM in design time and verification at runtime than the previous model checking tools. Moreover, the experimental results demonstrate that the proposed approach is better at path finding (i.e., reachability) than the other modeling tools (i.e., Cadence SMV, NuSMV, nuXmv, and RINGA). Additionally, we present a discussion on the limitations of the proposed cache mechanism, and future research is included to address these limitations. In future research, the proposed approach will be enhanced to solve the limitations.

References

[1] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[2] E. A. Emerson, "The beginning of model checking: A personal perspective," in *25 Years of Model Checking*. Berlin, Germany: Springer, 2008, pp. 27–45.

[3] N. Evans, "Verifying QThreads: Is model checking viable for user level tasking runtimes?" in *Proc. IEEE/ACM 2nd Int. Workshop Softw. Correctness HPC Appl. (Correctness)*, 2018, pp. 25–32.

[4] U. Vogl and M. Siegle, "A new approach to predicting reliable project runtimes via probabilistic model checking," in *Proc. Eur. Workshop Perform. Eng.*, 2017, pp. 117–132.

[5] A. Desai, T. Dreossi, and S. A. Seshia, "Combining model checking and runtime verification for safe robotics," in *Proc. Int. Conf. Runtime Verification*, 2017, pp. 172–189.

[6] X. Zhao, V. Robu, D. Flynn, F. Dinmohammadi, M. Fisher, and M. Webster, "Probabilistic model checking of robots deployed in extreme environments," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 8066–8074.

[7] K. Kejstová, P. Ročkai, and J. Barnat, "From model checking to runtime verification and back," in *Proc. Int. Conf. Runtime Verification*, 2017, pp. 225–240.

[8] V. C. Ngo, A. Legay, and J. Quilbeuf, "Statistical model checking for systemC models," in *Proc. IEEE 17th Int. Symp. High Assurance Syst. Eng. (HASE)*, 2016, pp. 197–204.

[9] E. Lee, Y.-G. Kim, Y.-D. Seo, and D.-K. Baik, "Self-adaptive framework with game theoretic decision making for Internet of Things," in *Proc. IEEE TENCON Region 10 Conf.*, 2018, pp. 2092–2097.

[10] E. Lee, Y.-D. Seo, and Y.-G. Kim, "Self-adaptive framework based on MAPE loop for Internet of Things," *Sensors*, vol. 19, no. 13, p. 2996, 2019.

[11] A. Naskos, E. Stachtiari, P. Katsaros, and A. Gounaris, "Probabilistic model checking at runtime for the provisioning of cloud resources," in *Runtime Verification*. Cham, Switzerland: Springer, 2015, pp. 275–280.

[12] Y. Zhao and F. Rammig, "Online model checking for dependable real-time systems," in *Proc. IEEE 15th Int. Symp. Object Component Service Orient. Real Time Distrib. Comput.*, 2012, pp. 154–161.

[13] M. Qanadilo, S. Samara, and Y. Zhao, "Accelerating online model checking," in *Proc. IEEE 6th Latin–Amer. Symp. Depend. Comput.*, 2013, pp. 40–47.

[14] K. Sudhakar, Y. Zhao, and F.-J. Rammig, "Efficient integration of online model checking into a small-footprint real-time operating system," *Concurrency Comput. Pract. Exp.*, vol. 28, no. 14, pp. 3773–3797, 2016.

[15] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D.-K. Baik, "Runtime verification method for self-adaptive software using reachability of transition system model," in *Proc. ACM Symp. Appl. Comput.*, 2017, pp. 65–68.

[16] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D. Baik, "RINGA: Design and verification of finite state machine for self-adaptive software at runtime," *Inf. Softw. Technol.*, vol. 93, pp. 200–222, Jan. 2018.

[17] H. Nakagawa, H. Toyama, and T. Tsuchiya, "Expression caching for runtime verification based on parameterized probabilistic models," *J. Syst. Softw.*, vol. 156, pp. 300–311, Oct. 2019.

[18] A. Filieri, G. Tamburrelli, and C. Ghezzi, "Supporting self-adaptation via quantitative verification and sensitivity analysis at run time," *IEEE Trans. Softw. Eng.*, vol. 42, no. 1, pp. 75–99, Jan. 2016.

[19] D. Weyns and U. Iftikhar, "Model-based simulation at runtime for self-adaptive systems," in *Proc. Models Runtime Würzburg*, 2016, pp. 1–9.

[20] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, 2010.

[21] *Information Technology—Internet of Things (IoT)—Vocabulary*, Int. Org. Stand., Geneva, Switzerland, Dec. 2018.

[22] L. Liang, K. Zheng, Z. Wei, Y. Wang, S. Wu, and X. Huang, "Model checking of IoT system in microgrid," in *Proc. IEEE 8th Int. Conf. Inf. Technol. Med. Educ. (ITME)*, 2016, pp. 601–605.

[23] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic model checking for performance and reliability analysis," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 4, pp. 40–45, 2009.

[24] B. Costa, P. F. Pires, F. C. Delicato, W. Li, and A. Y. Zomaya, "Design and analysis of IoT applications: A model-driven approach," in *Proc. IEEE 14th Int. Conf. Depend. Auton. Secure Comput. 14th Int. Conf. Pervasive Intell. Comput. 2nd Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Congr. (DASC/PiCom/DataCom/CyberSciTech)*, 2016, pp. 392–399.

[25] T. Kuroiwa, Y. Aoyama, and N. Kushiro, "A hybrid testing environment between execution test and model checking for IoT system," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, 2019, pp. 1–2.

[26] M. Mohsin, M. U. Sardar, O. Hasan, and Z. Anwar, "IoTRiskAnalyzer: A Probabilistic model checking based framework for formal risk analytics of the Internet of Things," *IEEE Access*, vol. 5, pp. 5494–5505, 2017.

[27] L. Li, Z. Jin, G. Li, L. Zheng, and Q. Wei, "Modeling and analyzing the reliability and cost of service composition in the IoT: A probabilistic approach," in *Proc. IEEE 19th Int. Conf. Web Services*, 2012, pp. 584–591.

[28] S. Yamaguchi, S. Tsugawa, and K. Nakahori, "An analysis system of IoT services based on agent-oriented Petri net PN2," in *Proc. IEEE Int. Conf. Consum. Electron. Taiwan (ICCE-TW)*, 2016, pp. 1–2.

[29] K. Nakahori and S. Yamaguchi, "A support tool to design IoT services with NUSMV," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, 2017, pp. 80–83.

[30] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "IoTSAN: Fortifying the safety of IoT systems," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, 2018, pp. 191–203.

[31] S. Xu, W. Miao, T. Kunz, T. Wei, and M. Chen, "Quantitative analysis of variation-aware Internet of Things designs using statistical model checking," in *Proc. IEEE Int. Conf. Softw. Qual. Rel. Security (QRS)*, 2016, pp. 274–285.

[32] A. Rodríguez, L. M. Kristensen, and A. Rutle, "Formal modelling and incremental verification of the MQTT IoT protocol," in *Transactions on Petri Nets and Other Models of Concurrency XIV*. Berlin, Germany: Springer, 2019, pp. 126–145.

[33] K. Keerthi, I. Roy, A. Hazra, and C. Rebeiro, "Formal verification for security in IoT devices," in *Security and Fault Tolerance in Internet of Things*. Cham, Switzerland: Springer, 2019, pp. 179–200.

[34] S. Ouchani, "Ensuring the functional correctness of IoT through formal modeling and verification," in *Proc. Int. Conf. Model Data Eng.*, 2018, pp. 401–417.

[35] *The NUXMV Model Checker*. Accessed: Sep. 9, 2019. [Online]. Available: https://nuxmv.fbk.eu

[36] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, "Model-checking algorithms for continuous-time Markov chains," *IEEE Trans. Softw. Eng.*, vol. 29, no. 6, pp. 524–541, Jun. 2003.

[37] G. Su, T. Chen, Y. Feng, and D. S. Rosenblum, "ProEva: runtime proactive performance evaluation based on continuous-time Markov chains," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 484–495.

[38] E. Lee, Y.-D. Seo, and Y.-G. Kim, "A Nash equilibrium based decision-making method for Internet of Things," *J. Ambient Intell. Humanized Comput.*, to be published.

[39] K. L. McMillan, *The SMV Language*, Cadence Berkeley Labs, San Jose, CA, USA, pp. 1–49, 1999.

[40] A. Cimatti *et al.*, "NUSMV2: An opensource tool for symbolic model checking," in *Proc. Int. Conf. Comput.-Aided Verification*, 2002, pp. 359–364.

[41] R. Cavada *et al.*, "The NUXMV symbolic model checker," in *Proc. Int. Conf. Comput.-Aided Verification*, 2014, pp. 334–342.

[42] E. M. Clarke, "The birth of model checking," in *25 Years of Model Checking*. Berlin, Germany: Springer, 2008, pp. 1–26.

[43] H. Nakagawa, K. Ogawa, and T. Tsuchiya, "Caching strategies for run-time probabilistic model checking." in *Proc. MoDELS Run Time*, 2016, pp. 18–25.

[44] A. Filieri *et al.*, "Software engineering meets control theory," in *Proc. 10th Int. Symp. Softw. Eng. Adapt. Self Manag. Syst.*, 2015, pp. 71–82.

[45] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proc. ACM 33rd Int. Conf. Softw. Eng.*, 2011, pp. 341–350.

[46] A. Filieri and G. Tamburrelli, "Probabilistic verification at runtime for self-adaptive systems," in *Assurances for Self-Adaptive Systems*. Berlin, Germany: Springer, 2013, pp. 30–59.

[47] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proc. 16th Int. ACM SIGSOFT Symp. Compon. Softw. Eng.*, 2013, pp. 33–42.

[48] E. A. Emerson and J. Y. Halpern, "sometimes and not never revisited: On branching versus linear time temporal logic," *J. ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[49] T. Wilke, "CTL+ is exponentially more succinct than CTL," in *Proc. Int. Conf. Found. Softw. Technol. Theor. Comput. Sci.*, 1999, pp. 110–121.

[50] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects Comput.*, vol. 6, no. 5, pp. 512–535, 1994.

[51] E. M. Hahn, T. Han, and L. Zhang, "Synthesis for PCTL in parametric Markov decision processes," in *Proc. NASA Formal Methods Symp.*, 2011, pp. 146–161.

[52] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Verifying continuous time Markov chains," in *Proc. Int. Conf. Comput.-Aided Verification*, 1996, pp. 269–276.

[53] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York, NY, USA: Springer, 2012.

[54] J.-P. Katoen, *Concepts, Algorithms, and Tools for Model Checking*. IMMD, Erlangen, Germany, 1999.

[55] G. Agha, J. Meseguer, and K. Sen, "PMAUDE: Rewrite-based specification language for probabilistic object systems," *Electron. Notes Theor. Comput. Sci.*, vol. 153, no. 2, pp. 213–239, 2006.

[56] S. Sebastio and A. Vandin, "MultiVesta: Statistical model checking for discrete event simulators," in *Proc. 7th Int. Conf. Perform. Eval. Methodol. Tools*, 2013, pp. 310–315.

[57] B. Bonakdarpour and B. Finkbeiner, "Runtime verification for HyperLTL," in *Proc. Int. Conf. Runtime Verification*, 2016, pp. 41–45.

[58] J. Cámara, D. Garlan, B. Schmerl, and A. Pandey, "Optimal planning for architecture-based self-adaptation via model checking of stochastic games," in *Proc. 30th Annu. ACM Symp. Appl. Comput.*, 2015, pp. 428–435.

[59] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *Proc. ICSE Workshop Softw. Eng. Adapt. Self Manag. Syst.*, 2009, pp. 132–141.

[60] B. Schmerl *et al.*, "Architecture-based self-protection: composing and reasoning about denial-of-service mitigations," in *Proc. ACM Symp. Bootcamp Sci. Security*, 2014, p. 2.

[61] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for NOSQL clusters using TIRAMOLA," in *Proc. 13th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput.*, 2013, pp. 34–41.

[62] Q. Qiu, Q. Qu, and M. Pedram, "Stochastic modeling of a power-managed system-construction and optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 10, pp. 1200–1217, Oct. 2001.

[63] H. Gao, H. Miao, and H. Zeng, "Service reconfiguration architecture based on probabilistic modeling checking," in *Proc. IEEE Int. Conf. Web Services*, 2014, pp. 714–715.

[64] J. Barnat *et al.*, "Divine 3.0—An explicit-state model checker for multithreaded C & C++ programs," in *Proc. Int. Conf. Comput.-Aided Verification*, 2013, pp. 863–868.

[65] B. Boyer, K. Corre, A. Legay, and S. Sedwards, "Plasma-lab: A flexible, distributable statistical model checking library," in *Proc. Int. Conf. Quant. Eval. Syst.*, 2013, pp. 160–164.

[66] T. Sekizawa, T. Mikoshi, M. Nagura, R. Watanabe, and Q. Chen, "Probabilistic position estimation and model checking for resource-constrained IoT devices," in *Proc. 27th Int. Conf. Comput. Commun. Netw. (ICCCN)*, 2018, pp. 1–7.

[67] *Verifying Multi-Threaded Software With Spin*. Accessed: Jan. 3, 2020. [Online]. Available: http://spinroot.com/

[68] *ISO/IEC/IEEE International Draft Standard—Systems and Software Engineering—Guide for the Utilization of ISO/IEC/IEEE 15288 in the Context of System of Systems Engineering*, ISO/IEC/IEEE Standard P21840, pp. 1–64, Apr. 2019.

[69] *NUSMV: A New Symbolic Model Checker*. Accessed: Sep. 8, 2019. [Online]. Available: http://nusmv.fbk.eu/

[70] *OASIS Standard MQTT Version 3.1.1*. Accessed: Sep. 9, 2019. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/csprd02/mqtt-v3.1.1-csprd02.html

[71] M. Houimli, L. Kahloul, and S. Benaoun, "Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things," in *Proc. IEEE Int. Conf. Math. Inf. Technol. (ICMIT)*, 2017, pp. 214–221.

[72] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: A language and code generation framework for heterogeneous targets," in *Proc. ACM/IEEE 19th Int. Conf. Model Driven Eng. Lang. Syst.*, 2016, pp. 125–135.

[73] F. Fleurey and B. Morin, "ThingML: A generative approach to engineer heterogeneous and distributed systems," in *Proc. IEEE Int. Conf. Softw. Archit. Workshops (ICSAW)*, 2017, pp. 185–188.

[74] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 321–332, 2013.

[75] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2004, pp. 168–176.

[76] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Autonomous Adapt. Syst.*, vol. 4, no. 2, p. 14, 2009.

[77] R. De Lemos *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Berlin, Germany: Springer, 2013, pp. 1–32.

[78] A. Rayes and S. Salam, *Internet of Things—From Hype to Reality*. Cham, Switzerland: Springer, 2017.

[79] A. Ghosh, D. Chakraborty, and A. Law, "Artificial intelligence in Internet of Things," *CAAI Trans. Intell. Technol.*, vol. 3, no. 4, pp. 208–218, 2018.

[80] G. Fortino, R. Gravina, W. Russo, and C. Savaglio, "Modeling and simulating Internet-of-Things systems: A hybrid agent-oriented approach," *Comput. Sci. Eng.*, vol. 19, no. 5, pp. 68–76, 2017.

[81] G. Fortino, W. Russo, C. Savaglio, W. Shen, and M. Zhou, "Agent-oriented cooperative smart objects: From iot system design to implementation," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 48, no. 11, pp. 1939–1956, Nov. 2018.

[82] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, and M. Viroli, "Modelling and simulation of opportunistic IoT services with aggregate computing," *Future Gener. Comput. Syst.*, vol. 91, pp. 252–262, Feb. 2019.

Euijong Lee received the B.S. degree in computer information and science and the Ph.D. degree in computer science and engineering from Korea University, Seoul, South Korea, in 2012 and 2018, respectively.

He is currently a Postdoctoral Researcher with the Department of Computer and Information Security, Sejong University, Seoul. His research interests include self-adaptive software, software engineering, model checking, Internet of Things, and data mining.

Young-Duk Seo received the B.S. degree in computer and communication engineering and the Ph.D. degree in computer science and engineering from Korea University, Seoul, South Korea, in 2012 and 2018, respectively.

He was a Research Professor with the Computer, Information and Communication Research Institute, Korea University and a Postdoctoral Researcher with the Department of Computer and Information Security, Sejong University, Seoul, where he is currently an Assistant Professor with the Department of Data Science. His research interests include self-adaptive software, big data analysis, recommender system, and entity linking.

Young-Gab Kim (Member, IEEE) received the B.S. degree in biotechnology and genetic engineering and minored in computer science and engineering and the M.S. and Ph.D. degrees in computer science and engineering from Korea University, Seoul, South Korea, in 2001, 2003, and 2006, respectively.

He was an Assistant Professor with the School of Information Technology, Catholic University of Daegu, Gyeongsan, South Korea. He is currently an Associate Professor with the Department of Computer and Information Security, Sejong University, Seoul. He has published over 130 research papers in the field of computer science and information security. His current research interests include big data security, network security, home network, security risk analysis, and security engineering.

Dr. Kim is contributing in developing data exchange standards as a Korean ISO/IEC JTC 1 Member.