Self-Adaptive Framework With Master–Slave Architecture for Internet of Things

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

***Abstract*—The Internet of Things (IoT) connects a wide range of entities and can be applied to various types of environments. In addition, IoT environments can be dynamically changed at runtime; thus, IoT systems can be deployed in various environments. To support stable operation, IoT systems must adapt to dynamic environmental changes. The self-adaptive software aims to adjust various artifacts or attributes of software to adapt the detected context by itself, and various studies have applied self-adaptive methods in IoT-related research. In this study, we proposed a self-adaptive software framework with master–slave architecture-based finite-state machine modeling. In addition, model checking is applied, which is a formal method to verify IoT systems at runtime, and a cache-based mechanism is applied to reduce the computational time required for verification. To demonstrate the efficiency of the proposed framework, an empirical evaluation was performed with several model-checking tools (RINGA, NuSMV, nuXmv, and CadenceSMV), and the results showed the efficiency of the proposed framework with the cache-based mechanism. In addition, an example application was investigated with smart greenhouse scenarios, and the application was implemented on Android and Arduino. The application was operated in physical environments, and the results showed the practical usability of the proposed framework with verification at runtime.**

***Index Terms*—Finite-state machine, Internet of Things (IoT), model checking, self-adaptive software.**

I. Introduction

THE Internet of Things (IoT) interconnects various entities called “things” having a distinct existence including physical or nonphysical (e.g., sensor, actuator, digital user, and human user) [1], [2]. Therefore, IoT systems can support the collection of data from sensor-related entities, change physical-related actions by actuator-related entities, and generate meaningful information by analyzing the collected data from the entities. With these characteristics, various IoT-related studies have focused on various domains (e.g., drones [3], [4], smart buildings [5], [6], video surveillance [7], and health care applications [8]). In addition, IoT systems

Manuscript received 8 September 2021; revised 16 December 2021; accepted 7 February 2022. Date of publication 10 February 2022; date of current version 24 August 2022. This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIT) under Grant 2021R1A2C2012635 and Grant 2021R1G1A101097111. *(Corresponding author: Young-Gab Kim.)*

Euijong Lee is with the School of Computer Science, Chungbuk National University, Cheongju 28644, South Korea (e-mail: kongjjagae@cbnu.ac.kr).

Young-Duk Seo is with the Department of Computer Engineering, Inha University, Incheon 22212, South Korea (e-mail: mysid88@inha.ac.kr).

Young-Gab Kim is with the Department of Computer and Information Security, and Convergence Engineering for Intelligent Drone, Sejong University, Seoul 05006, South Korea (e-mail: alwaysgabi@sejong.ac.kr).

Digital Object Identifier 10.1109/JIOT.2022.3150598

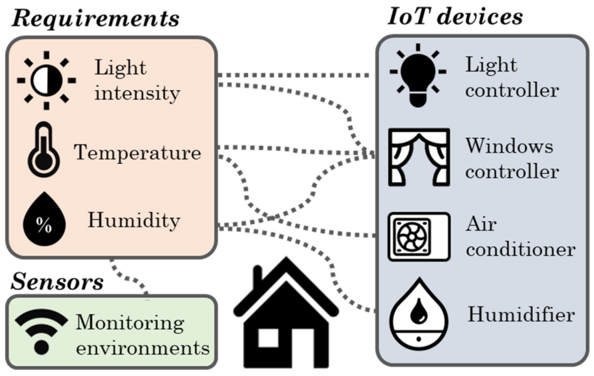


Fig. 1. Example of a dynamic IoT environment.

can be dynamically changed for various reasons, such as changing user requirements, joining new IoT devices, losing IoT devices, or occurring environmental changes. Therefore, IoT systems in dynamic environments must support stable execution at runtime to adapt to various changes.

The self-adaptive software aims to adapt detected contexts that denote all aspects in an environment that affect software operation by adjusting various artifacts or attributes of the software. Thus, one of the important goals of the self-adaptive software is to support stable execution of the software under dynamic changes, including environmental and software changes. In this context, various studies on self-adaptation can be applied to IoT systems to support stable execution with the minimized human involvement. Recently, the self-adaptive software has been applied to IoT systems from various viewpoints and IoT domains [3], [5]–[23].

Several IoT systems can be dynamically changed for various reasons, such as changing user requirements, joining new IoT devices, losing IoT devices, or environmental changes. The motivation of this study is to apply self-adaptive concepts to IoT systems with dynamically changing environments to support stable execution at runtime. In addition, we focused on IoT environments in which IoT devices and requirements are closely related; thus, the operations of the actuators can affect sensed values and requirement satisfaction. Fig. 1 shows an IoT-based smart home example that we focused on.

In the example, there are three requirements (i.e., light intensity, temperature, and humidity) with sensors and four IoT devices (i.e., light controller, windows controller, air conditioner, and humidifier). It is assumed that several sensors monitor the environmental factors related to requirements.

The dotted line denotes the relationships between the requirements and the IoT devices. As depicted in the figure, each requirement is related to several devices, and this denotes that an operation can affect multiple requirements (e.g., the temperature requirement can be changed by the operation of two devices). Although this is a simple example, it involves various dynamic factors: 1) IoT devices can be connected or disconnected; 2) the requirements satisfaction criteria can be changed; 3) the operations of IoT devices affect related requirements (e.g., turning on the air conditioner to reduce temperature can affect humidity); and 4) external environmental factors affect requirements (e.g., the requirements can be affected by outside conditions). In addition, a smart home requires self-adaptation to satisfy the requirements of dynamic changes. In addition to this example, there are several IoT devices that operate in dynamic environments. In addition, recent studies have applied self-adaptation in various IoT domains (e.g., emergency handling systems [21], video surveillance [7], smart building [6], automated parking lots [17], health IoT [8], and greenhouses [22]).

In this study, we proposed a self-adaptive framework to support dynamic adaptation in IoT environments. The proposed approach is based on the initial study (i.e., RINGA [9]), which is a self-adaptive software framework with model checking for verification at runtime. The initial study includes finite-state machine modeling for self-adaptive software, and it is expended for modeling IoT environments [12]. However, there are some limitations in previous research: 1) increasing computing power is required when the model size is large and complicated and 2) the modeling can be performed in a host device; thus, the relationship among IoT entities is predefined.

To accomplish the motivation and overcome the limitations, we proposed a framework based on the concept of a self-adaptive system; the main system features and contributions of our work are summarized as follows.

1) An enhanced finite-state machine was proposed to describe IoT systems with a master/slave architecture to enhance the configuration between various IoT artifacts.
2) A self-adaptive process was proposed using the proposed finite-state machine. In the adaptation process, the finite-state machine was used for verification at runtime using the cache-based abstraction method [24], and the game-theory-based algorithm [11] was applied to determine adaptation strategies. The applied methods are in good agreement with the proposed finite-state machine in the self-adaptive process.
3) A comparison with our earlier work (i.e., RINGA [10]) and model checkers (i.e., CadenceSMV [25], NuSMV [26], [27], and nuXmv [28], [29]) were performed. Reasonable results were obtained, indicating that the proposed approach can be applied at runtime in complex IoT environments.
4) An IoT-based greenhouse application was implemented in the real physical world to demonstrate the usability and effectiveness of the proposed framework.

The remainder of this article is organized as follows. Section II provides the background and related work. Section III introduces the proposed self-adaptive framework for the IoT environment. Section IV presents the results of empirical experiments compared with other model-checking tools. Section V presents the results of experiments with an IoT-based smart greenhouse application. Section VI discusses the limitations and future work, and Section VII concludes the study.

II. Background and Related Work

In this section, the background and related work are described. Model checking is introduced in Section II-A. Section II-B briefly describes the definition of the self-adaptive software. Section II-C describes the self-adaptive software with the abstraction-based model-checking method, which is the initial research aim of this study. In Section II-D, self-adaptive software-based IoT studies are introduced.

A. Model Checking

Model checking is a verification technique that uses a transition system and temporal logic [30]. The transition system is used to model target systems, and temporal logic is used to describe specifications (i.e., detailed requirements of the designed model). The transition system consists of states and transitions between states. The status denotes the specific status of the designed system, and transitions describe the directed transition between states. The transition system used in model checking can be described as follows [30].

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.

In addition, the transition system can be used in various forms, such as finite-state machine [9]–[12], [24], discrete-time Markov chain [31]–[34], timed automata [35], and probabilistic models [36], [37]. As described, temporal logics are described specifications in model checking, and linear time logic (LTL) and computation tree logic (CTL) are generally used to describe specifications [38]. CTL is a branching-time logic; thus, CTL describes specifications in which the future is not determined.

In addition, LTL and CTL have been improved by several studies to describe more expressions [39]–[49]. In this study, we applied model checking to verify IoT systems, and the model abstraction approach was used to enhance the verification performance at runtime. The details are presented in the next section.

B. Self-Adaptive Software

The self-adaptive software aims to adjust various artifacts or attributes of software to adapt the detected context by itself [50]. The context denotes everything in environments that can affect the software. In addition, for the self-adaptive software, detecting the context and adapting operations are continuously required; thus, a cycle is needed to deal with self-adaptation [50]. The cycle is called the adaptation loop, and the loop consists of four processes. The processes are described as follows.

1) The monitoring process is responsible for collecting data from the software itself and the operating environment.
2) The analysis (detection) process is responsible for analyzing the symptoms using data from the monitoring process.
3) The planning (deciding) process is responsible for deciding how to change artifacts or attributes to achieve better performance (i.e., it is responsible for detecting adaptation strategies if adaptation is required).
4) The executing (acting) process is responsible for applying the change (i.e., adaptive strategy).

The loop that consists of four processes is called the MAPE loop. In addition, the loop can share knowledge to share data among the processes; this loop is called the MAPE-K loop [51]. However, the concept of the self-adaptive software has been applied to various IoT studies [3], [5]–[8], [13]–[21], [52]–[56]. The details of the application of the self-adaptive IoT are described in Section II-D. In addition, the proposed framework consists of the MAPE-K loop for IoT systems, and the details are described in Section III.

C. Self-Adaptive Software With Model Checking

The proposed framework is based on the self-adaptive software and a model abstraction-based verification method. The initial studies are introduced in this section. First, the self-adaptive software framework with model checking is introduced in Section II-C1. The cache-based model abstraction is presented in Section II-C2. In Section II-C3, a self-adaptive software framework for IoT is introduced. The limitations of the initial studies and the purpose of the proposed framework are described in each section.

1) Self-Adaptive Software Framework With Model Abstraction: In this section, a model abstraction-based self-adaptive software framework is introduced and the framework is called RINGA [10]. RINGA was proposed to design the self-adaptive software with model checking. To apply model checking, RINGA provided rules to design a finite-state machine and a finite-state machine called self-adaptive FSM (i.e., SA-FSM). SA-FSM reflected the characteristics of the self-adaptive software, and the finite-state machine was used to design a self-adaptive system.

RINGA applies model checking to verify the system at runtime, but model checking has a chronic problem called state explosion, in which, the number of state spaces exponentially increases with verification time; thus, the problem interrupts verification using model checking at runtime [30]. To solve this problem, various techniques have been developed [38], and a model abstraction method was proposed in RINGA. However, RINGA has limitations, the first of which is related to the design of a finite-state machine. SA-FSM provided rules to design a finite-state machine to describe the self-adaptive software, but the design process requires assumptions on how to adapt to environmental changes. Therefore, if an unexpected environmental condition occurs, the finite-state machine must be redesigned.

2) Cache-Based Model Abstraction: A method to enhance the finite-state machine abstraction is introduced in this section. RINGA has limitations in terms of the abstraction performance of finite-state machines. The limitation is closely related to computing power; thus, applying RINGA to IoT devices with low computing power is challenging. To overcome this limitation, cache-based model abstraction has been proposed [24]. The cache-based method significantly reduced not only the abstraction time but also the verification time. Additionally, the cache-based method required less computing power than RINGA; thus, the method was applied in this study to verify IoT systems at runtime.

3) Self-Adaptive Software Framework With Model Abstraction for IoT: To apply self-adaptive software concepts to IoT systems, a RINGA-based self-adaptive framework was proposed [12], and the framework is called RINGA-IoT for convenience in this study. A finite-state machine was proposed in RINGA-IoT, and the finite-state machine was based on SA-FSM. The finite-state machine was simplified by the abstraction method in RINGA, and the abstracted model was used to verify the designed IoT system. Additionally, RINGA-IoT applied Nash equilibrium-based strategy extraction methods [11] to extract adaptive solutions. RINGA-IoT provides rules to construct a finite-state machine to design an IoT system with self-adaptive software concepts, but it has some limitations. The finite-state machine in RINGA-IoT has limitations in describing complex relations. In addition, RINGA-IoT abstracts the finite-state machine using the abstraction method in RINGA; thus, it has the same limitation that occurs in RINGA (i.e., computing power is required to abstract complex models). Several IoT devices may have limited computing power; thus, reducing computing power is required.

D. Self-Adaptive Software in IoT

In this section, recent studies that focus on self-adaptation for IoT-based systems and various research topics (e.g., architecture, algorithm, system, or framework) are introduced. Various IoT studies focusing on software-based approaches for self-adaptive systems have been conducted. Andrade *et al.* [5] proposed a self-adaptive IoT infrastructure to support various facets, and they focused on 1) contextual discovery of smart objects; 2) context awareness and managers; and 3) the self-adaptation process. The infrastructure consists of existing solutions (i.e., CoAP-CTX [57], LoCCAM [58], and SUUCCEEd [59]). The framework follows the MAPE-K loop (i.e., control, verification, and execution). In the infrastructure, LoCCAM is used to discover IoT objects and to manage context. Finally, SUCCEEd is used to control adaptation in IoT systems by considering the context. As proof of concept, the infrastructure applies smart buildings with embedded sensors and actuators for efficient use of resources, detection of emergencies, and optimization of processes. A comprehensive architecture was proposed by Serhani *et al.* [8] and an architecture to integrate IoT workflow specification, orchestration, monitoring, prediction, and adaptation was developed. The IoT workflow denotes processes to handle IoT data, including the collection, analysis, and automated decision processes. Therefore, the architecture focuses on workflow interoperability, Quality-of-Service (QoS) requirements, workflow orchestration, and execution with IoT data in the cloud infrastructure. Various tools, programming languages, and application programming interfaces (APIs) have been introduced that may be used in cloud-based IoT environments. In addition, a health IoT example was applied to demonstrate how the architecture operated with workflows. In this example, adaptation algorithms are introduced to show how workflows are automated and optimized in the proposed architecture. However, the architecture mainly focuses on describing the orchestration of workflow; thus, general algorithms or methods for self-adaptation are insufficient. Muccini *et al.* [21] presented an IoT distribution pattern with self-adaptation, and the patterns and self-adaptation were combined with respect to their specific characteristics. In addition, an IoT modeling framework for an emergency handling system based on the MAPE loop is proposed, and the framework is simulated with an IoT-based forest monitoring system. Ariza *et al.* [52] proposed an adaptive IoT architecture for managing dynamic adaptation. The architecture focuses on two challenges: 1) resiliency to transient IoT devices and 2) the inclusion of new IoT devices. To solve these challenges, ontology is applied to represent domain knowledge (i.e., definitions of services and devices). In addition, matching and update algorithms have been proposed to support device management. The architecture is applied in a sustainable urban drainage system case study that monitors the reduction of water in nature to prevent negative impacts in urban areas. A goal-driven architecture and a MAPE-K loop-based process to deploy IoT nodes in Edge-Cloud environments have been proposed [6]. In the architecture, the goals are described as ontology, and the self-adaptive process supports deployments of the IoT system to achieve user goals (i.e., requirements from users). The goal-driven architecture was simulated in a smart home and smart building scenarios. An IoT network architecture was proposed using an adaptive control loop and blockchain [13]. The architecture focuses on the block flow between IoT devices and the blockchain. A self-adaptive control algorithm was proposed to improve the efficiency, and the loop consisted of monitoring and control processes. Shin *et al.* [14] developed a dynamic adaptive software-defined network (SDN) configuration approach for IoT environments, and the approach was named DICES. The DICES consists of a feedback loop (i.e., monitor, analyze, compute, and apply). In addition, a search algorithm was proposed to minimize network link utilization, reconfiguration cost, and transmission. DICES was simulated in an IoT-enabled national emergency management system, and the results showed an efficiency of adaptation to resolve congestion.

In addition, hardware-based approaches also exist for IoT technologies. Burger *et al.* [3] proposed an IoT framework to support the development and deployment of distributed IoT environments with adaptive hardware for continuous change, and the framework was called the elastic IoT platform. The elastic IoT platform focused on IoT hardware-based adaptation, and the embedded hardware was modeled using a previous study (i.e., elastic node [53]). Based on the elastic node, various factors (e.g., interaction scheme, computing power, or data) were provided as a service, and the platform provided flexibility and convenience to the hardware layer in the edge and cloud using the provided factors. In addition, a self-aware drone system was applied to show an efficient elastic IoT platform. An access control middleware [20] was presented for IoT, and the middleware was capable of adapting changes. The proposed middleware has a dynamic adaptation process of access control rules to satisfy the requirements of the IoT environment. A hardware and software architecture for designing dynamic reconfigurable IoT environments was presented [15]. The architecture aims to remotely controllable IoT nodes that are connected to sensor processing in runtime with adaptation. The architecture presents the node architecture, and the nodes are dynamically reconfigured in an adaptive runtime manager (ADAM). The ADAM provides an adaptive runtime management policy. With this policy, ADAM performs the configuration of hardware and software using messages from the designed nodes. The architecture was tested using an electrocardiogram (ECG) monitoring case.

Recently, various studies have applied machine learning for self-adaptation in the IoT. Nawaratne *et al.* [7] proposed self-evolving algorithms for data interoperability in IoT environments. They investigated the need for interoperable algorithms to support IoT environments, and the investigation used real data from the Metropolitan Fire Brigade (MFB) in Victoria, Australia. The algorithms focused on three requirements for IoT environments (i.e., unsupervised self-learning capability, self-generating to the environment, and incremental learning). To accomplish the self-evolving algorithms, unsupervised machine learning algorithms have been applied: growing self-organizing map (GSOM) [54] and incremental knowledge acquiring self-learning (IKASL) [55]. In addition, the self-evolving algorithms were evaluated with a video surveillance environment. Van Der Donckt *et al.* [16] presented a deep-learning-based approach for adaptation space reduction, and the approach was named DLASeR. DLASeR focused on the analysis process in the MAPE-K loop; thus, it focused on analyzing possible adaptation options and ranking the options. The DLASeR consists of two steps. The first step is to reduce the adaptation spaces to generate a threshold using a deep neural network (DNN), and the second step is to generate a rank to find the optimization goal using a regression DNN. DLASeR applies a self-adaptive exemplar that provides IoT network-related examples and is named DeltaIoT [56]. A MAPE-K loop-based self-adaptive architecture with the cooperation of machine learning and model checking has been proposed [17], [18]. To accomplish self-adaptation, reinforcement learning is applied to select adaptation patterns, and probabilistic model checking is used to verify the feasibility of the adaptation with respect to QoS requirements. In addition, the adaptation results are provided for reinforcement learning to improve future adaptations. The architecture was evaluated in scientific exhibition events, and its efficiency was demonstrated. Pauna *et al.* [19] presented a self-adaptive honeypot system to detect and prevent malicious attempts in

TABLE I
COMPARISON OF PREVIOUS SELF-ADAPTIVE RESEARCH

Worked by	**Year**	**Goal**	**Lifecycle**	**Approach**	**Experimental or simulation domain**
Muccini et al. [54]	2018	A framework for IoT modeling	MAPE loop	* Analysis IoT and self-adaptation control patterns * Bridge and combine IoT distribution patterns and adaptation logic	Emergency handling system
Nawaratne et al. [41]	2018	Self-evolving algorithms for data interoperability in an IoT environment	N/A	* GSOM based unsupervised self-organizing algorithm * IKASL based incremental learning algorithm	Video surveillance
Ouechtati et al. [53]	2018	A framework for access control in an IoT environment	Dynamic adaptation process (loop)	* Dynamic adaptive process based on risk value, policies, and rule sets	Middleware
Casado-Vara et al. [40]	2019	An architecture for a blockchain-based IoT network with adaptive closed-loop control	Closed-loop consisted of monitoring and control	* Self-adaptive algorithm with closed loop	N/A
Pauna et al. [52]	2019	A self-adaptive honey pot system in IoT environments	N/A	* Deep Q-learning to generate rewards for attackers	N/A
Scrugli et al. [46]	2019	An S/W and H/W architecture to design a dynamic reconfigurable IoT environment	N/A	* Node architecture * Adaptive policy	ECG monitoring use case
Alkhabbas et al. [49]	2020	A goal-driven architecture and process for deployments of goal-driven IoT system in Edge-Cloud	MAPE-K loop	* An architecture and elf-adaptation process to support deployment of an IoT system * Ontology-based goal description	Smart home and building
Burger et al. [32]	2020	A framework to support development and deploy IoT hardware	N/A	* IoT hardware modeling with elastic node * Resource-oriented approach to transform data and capabilities to available as-a-service	Self-aware drone system
Cámara et al. [50] and Muccini et al. [51]	2020	An architecture with machine learning and probabilistic model checking	MAPE-K loop	* Selecting adaptation patterns with reinforcement learning * Verification of adaptation feasibility with probabilistic model checking	Automated parking lots
Serhani et al. [39]	2020	An architecture for orchestration between different IoT workflows	N/A	* Integrating various IoT data (i.e., IoT workflow specification, orchestration, monitoring, prediction, and adaptation)	Health IoT example
Shin et al. [45]	2020	An approach to support a dynamic adaptive SDN configuration	feedback loop	* feedback-loop based configuration approach * search algorithm	Emergency management system
Van Der Donckt et al. [47]	2020	A deep learning-based adaptation space reduction approach	Analysis in MAPE-K loop	* DNN-based adaptation space reduction * Regression DNN-based ranking to find optimal solutions	DeltaIoT
Andrade et al. [34]	2021	An infrastructure to support development of self-adaptive system	MAPE-K loop	* Discover IoT objects using LoCCAM * Context awareness and managing with CoAP-CTX * Applying the Java-based framework (SUCCEEd) to develop self-adaptive system	Smart building
Ariza et al. [44]	2021	An adaptive IoT architecture to manage dynamic adaptation	N/A	* Matching & update algorithms * ontology-based description	Sustainable urban drainage system
Proposed	**2021**	**An IoT framework with cache-based model checking**	**MAPE-K loop**	*** IoT system modeling with master-slave architecture** *** Cache-based model checking method for runtime verification**	**Smart green house**

IoT environments. The honeypot system applied reinforcement learning, which is deep *Q*-learning, to generate long-term rewards for baiting attackers.

Each of these studies includes distinct characteristics with various approaches in different target IoT environments. Therefore, it is difficult to perform quantitative analysis; thus, we provide Table I for summarization and comparison to show the characteristics of the self-adaptive research for IoT. In this study, we focused on modeling IoT systems with a master/slave architecture based on RINGA [10]. In addition, cache-based model checking [24] is applied to enhance runtime verification performance. The details of the proposed approach are described in Section III.

III. SELF-ADAPTIVE FRAMEWORK WITH MASTER–SLAVE ARCHITECTURE FOR IOT

The initial research called RINGA [9], [10] provides a self-adaptive framework and is applied in IoT environments [12].

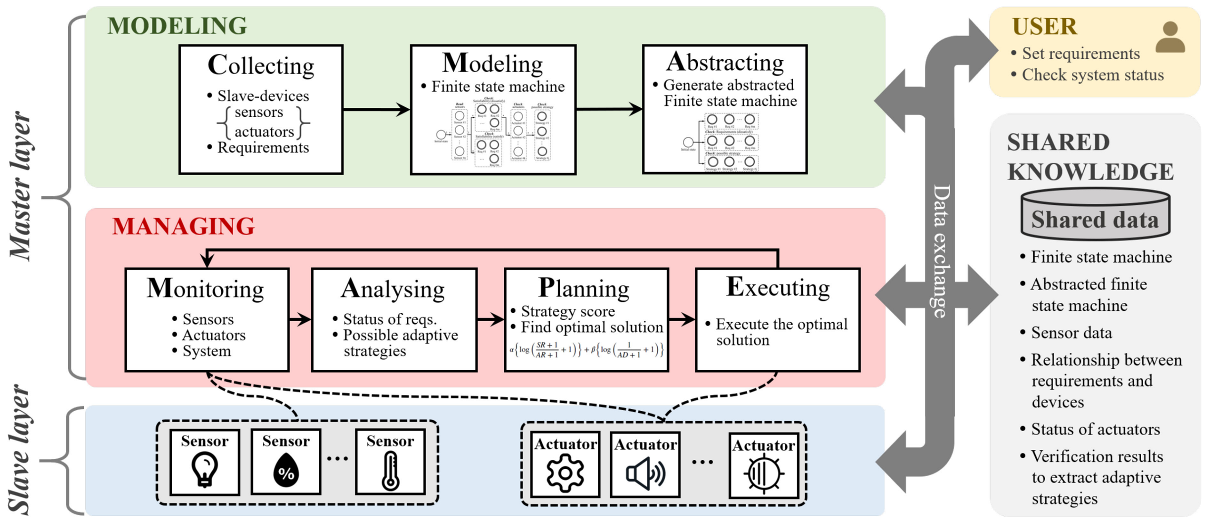


Fig. 2. Overview of the proposed framework with master–slave architecture.

However, previous research required a large amount of computing power, and thus, a cache-based model-checking method was employed to reduce the computing power for not only model abstraction but also runtime verification [24]. In addition, the previous IoT study [12] generates models that only reflect host-side views, and it has the same limitation as RINGA. To solve the limitations of previous research, we proposed a self-adaptive framework with a master–slave model [60] and a cache-based mechanism [24].

A. Overview

The proposed self-adaptive framework consists of two layers with shared knowledge. The layers consist of 1) a master layer and 2) a slave layer. Fig. 2 shows an overview of the proposed framework. First, shared knowledge (i.e., the right side of Fig. 2) saves and manages various factors that are used in the entire process of the framework. The details of the data that exist in the shared knowledge are described below with a description of the master and slave layers.

The user manages the overall IoT system and checks the system status using information from the shared knowledge. In addition, the user can provide additional information that cannot be extracted or deduced using the information from IoT devices. For example, if there is a sensor to detect environments (i.e., light intensity), but the requirements are not set, the user can set satisfiable ranges (i.e., 120–140 lux).

The slave layer consists of sensors and actuators. The sensor is an IoT device that measures one or more physical entities and outputs digital data that can be transmitted over a network [2]. The actuator is an IoT device that changes one or more properties of physical entities based on input values [2]. Both devices can only perform simple operations and do not operate alone; therefore, they require commends from the master layer to be operated harmoniously in an IoT system.

The master layer is responsible for modeling and managing; thus, the master executes the overall modeling and adaptation algorithms. Therefore, a device that performs operations in the master layer is a host device. The modeling phase consists of three parts: 1) collecting; 2) modeling; and 3) abstracting. The collecting process collects slave devices (i.e., sensors and actuators) and deduces the potential requirements from the collected sensors. The requirements are assumed to be determined by the existence of sensor devices because checking the status of the requirements requires more than one sensor. For example, if there is a light and humidity sensor, the potential requirement may be the humidity and light density of a physical area. However, user intervention is required to confirm the suitability of potential requirements. In addition, the collection process recognizes various relationships among the sensors, actuators, and requirements. Based on these relationships, a finite-state machine is generated to design the IoT system in the modeling process based on these relationships. The details of the finite-state machine are described in Section III-B. In the abstracting process, the finite-state machine describing the IoT system is abstracted as a simplified finite-state machine using cache-based abstracting [24] (see Section II-C2), and the simplified model is used to verify the IoT system at runtime. The reason for the abstraction is to prevent state explosion problems and enhance the verification performance at runtime. The data generated in the modeling phase are saved and managed in a shared database (i.e., the right side of Fig. 2), such as finite-state machines describing the IoT system, abstracted finite-state machines, sensors, actuators, requirements, and relationships.

The managing phase is responsible for supervising the IoT system at runtime. The managing phase consists of the MAPE loop, which is a prominent feedback loop used in the adaptation process in the self-adaptive software and autonomic computing [50]. The loop consists of four processes: 1) monitoring; 2) analysis; 3) planning; and 4) execution. The monitoring process is responsible for reading and updating the internal and external environments. Therefore, the process detects a change in the system's status and collects data from the slave devices. The monitored data are updated in the shared database and are used in other processes. The analysis process is responsible for verifying the status of requirements (i.e., each requirement is satisfied or unsatisfied) and to check

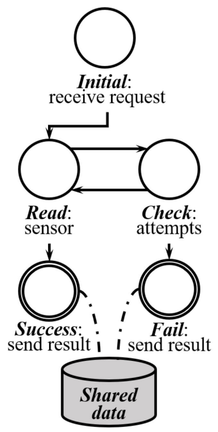


Fig. 3. Finite-state machine to design sensor device.

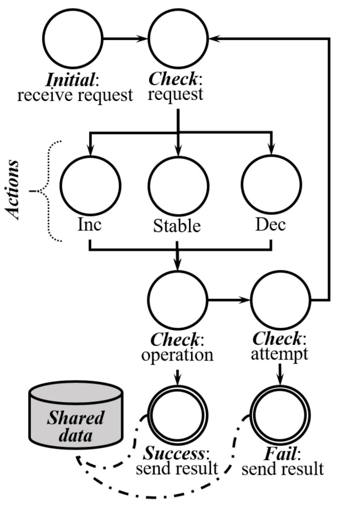


Fig. 4. Finite-state machine to design actuator.

possible adaptive strategies in a short time. Verification is performed with the abstracted finite-state machine that is generated in the modeling phase and cache mechanism [24]. In addition, shared data are used in this process, and the analyzed results are updated. The main purpose of the planning process is to find the optimal solution for adaptation. Therefore, if there are unsatisfied requirements, the process evaluates each possible adaptive strategy and selects the optimal solution using the strategy score method [11]. In addition, the optimal solution is saved in the shared data and is used in the executing process. The executing process is responsible for executing the actuators, and the executions are determined as the optimal solution. After the execution process, the loop continues, and the monitoring process is executed.

B. Finite-State Machine Design in Modeling Phase

In this section, the details of the finite-state machine for designing the master and slave layers are described. Finite-state machine modeling is conducted in the modeling phase; thus, this section describes the details of the modeling phase.

1) Finite-State Machine Model for the Slave Layer: The slave layer is responsible for the physical operation in IoT environments and systems. In international standards (i.e., ISO/IEC 20924:2018 Information technology—IoT—Vocabulary [2]), the term "IoT device" is defined as an entity of an IoT system that interacts and communicates with the physical world through sensing or actuating. Therefore, the slave layer is related to the IoT device design, and IoT devices consist of sensors and actuators [2]. As denoted by the standard, both devices can connect a network by themselves in this study.

The sensor device can measure the properties of one or more physical entities and generate outputs as digital data, and the result can be transmitted via a network [2]. Therefore, the input data of the sensor device are requests for sensing data, and the output is the sensor data if it is correctly operated. We proposed a general finite-state machine to design the sensor and named it SN-FSM. Fig. 3 shows SN-FSM, which can be expressed as a tuple (S, $\rightarrow$, S_{initial}, AP, L), where

1) S is a set of states;
2) S_{initial} is an initial state;
3) the states are classified into five types $\{S_{\text{initial}}, S_{\text{read}}, S_{\text{check}}, S_{\text{success}}, S_{\text{fail}}\} \subseteq S$;
4) S_{success} and S_{fail} are end states;
5) $\rightarrow \subseteq S \times S$ is the transition relation, and it is classified into five types $\{S_{\text{initial}} \times S_{\text{read}}, S_{\text{read}} \times S_{\text{success}}, S_{\text{read}} \times S_{\text{check}}, S_{\text{check}} \times S_{\text{read}}, S_{\text{check}} \times S_{\text{fail}}\}$;
6) AP is a set of atomic propositions;
7) $L : S \rightarrow 2^{\text{AP}}$ is a labeling function (2^{AP} denotes the power set of AP).

As indicated by the SN-FSM tuple definition, the finite-state machine compromises five states and five transitions. The state set and transitions include the following:

1) Initial state (S_{initial}) is an initial state.
2) The reading sensor (S_{raed}) reads the sensor properties. If the property is successfully read, the output is sent to the master layer ($S_{\text{read}} \times S_{\text{success}}$). However, if the reading sensor fails, it is transmitted to the checking attempt state ($S_{\text{read}} \times S_{\text{check}}$).
3) Checking attempt state S_{check} checks the attempts of the sensing property, and if the maximum attempts are not exceeded, it reaches the reading sensor again ($S_{\text{check}} \times S_{\text{read}}$). In addition, if the attempt is exceeded, it reaches the fail state ($S_{\text{check}} \times S_{\text{fail}}$).
4) Success S_{success} and fail S_{fail} states are the end states. The success state sends property values, and the failure state sends the failure report.

As depicted in Fig. 2, the results are saved in the shared database and used in various processes in the managing phase. Note that the SN-FSM is designed for general purposes to describe sensor devices as a finite-state machine; thus, it can be expanded by the different purposes of the sensor.

The actuator is an IoT device that changes one or more properties of the physical entity as a response to a valid input [2]. Therefore, an IoT system sends an operation as input to an actuator, and the actuator sends the results as output to the IoT system. A finite-state machine was proposed to design the actuator and was named ACT-FSM. Fig. 4 shows the ACT-FSM, which can be expressed as a tuple (S, $\rightarrow$, S_{initial}, AP, L), where

1) S is a set of states;

2) S_{initial} is an initial state;
3) the states are classified into nine types $\{S_{\text{initial}}, S_{\text{chkReq}}, S_{\text{inc}}, S_{\text{stable}}, S_{\text{dec}}, S_{\text{chkAct}}, S_{\text{chkAtmp}}, S_{\text{success}}, S_{\text{fail}}\} \subseteq S$;
4) S_{success} and S_{fail} are end states;
5) $\rightarrow \subseteq S \times S$ is the transition relation, and it is classified into five types $\{S_{\text{initial}} \times S_{\text{chkReq}}, S_{\text{cheReq}} \times \{S_{\text{inc}}, S_{\text{stable}}, S_{\text{dec}}\}, \{S_{\text{inc}}, S_{\text{stable}}, S_{\text{dec}}\} \times S_{\text{chkAct}}, S_{\text{chkAct}} \times \{S_{\text{chkAtmp}}, S_{\text{success}}\}, S_{\text{chkAtmp}} \times \{S_{\text{chkReq}}, S_{\text{fail}}\}\}$;
6) AP is a set of atomic propositions;
7) $L : S \rightarrow 2^{\text{AP}}$ is a labeling function (2^{AP} denotes the power set of AP).

Based on the ACT-FSM definition, the finite-state machine comprises 9 states and 11 transitions. The details of the states and transitions are described as follows.

1) Initial state (S_{initial}) is an initial state and receives the input value.
2) The checking request state (S_{chkReq}) verifies the input value, and the status is transited to the activation-related states by the input value ($S_{\text{cheReq}} \times \{S_{\text{inc}}, S_{\text{stable}}, S_{\text{dec}}\}$).
3) The activation-related states ($S_{\text{inc}}, S_{\text{stable}}, S_{\text{dec}}$) are related to the physical operation of an actuator; thus, the operations are executed in these states. After the operation status is reached to check the operation ($\{S_{\text{inc}}, S_{\text{stable}}, S_{\text{dec}}\} \times S_{\text{chkAct}}$). The purpose of the proposed finite-state machine is to provide a general design for the actuators. Therefore, the activations are only classified into three types of operations, but they can be expanded by the characteristics of an actuator.
4) The checking activation status (S_{chkAct}) verifies that the operations are executed correctly. The operation is executed without error, and the status reaches success status ($S_{\text{chkAct}} \times S_{\text{success}}$). However, if the operation is executed with error, the status is translated to check the attempt status ($S_{\text{chkAct}} \times S_{\text{chkAtmp}}$).
5) The checking attempts status counts the attempt of operations, and if the attempt exceeds a threshold, the status is translated to a failed state ($S_{\text{chkAtmp}} \times S_{\text{fail}}$). If the attempt does not exceed the threshold, the operation is retried ($S_{\text{chkAtmp}} \times S_{\text{chkReq}}$).
6) Success (S_{success}) and fail (S_{fail}) states are the end states. Both states send the results, and the results are saved in the shared data.

The finite-state machines in the slave layer describe each device in an IoT environment. In this study, it is assumed that IoT devices have finite-state machines to describe themselves, and the finite-state machines are sent to the master in the first pairing. In addition, finite-state machines are used in the master layer for system modeling and runtime verification.

2) Finite-State Machine Design for the Master Layer: The master layer is related to an IoT system that provides the functionalities of IoT [2]; thus, a finite-state machine is required to design overall IoT systems, check system status, and verify the system at runtime. To accomplish this, a finite-state machine is designed with the collected slave devices and requirements. The finite-state machine is named ML-FSM and can be expressed as a tuple (S, $\rightarrow$, S_{initial}, AP, L), where

1) S is a set of states;
2) S_{initial} is an initial state;
3) the states are classified into ten types: $\{S_{\text{initial}}, S_{\text{read}}, S_{\text{rSuccess}}, S_{\text{rFail}}, S_{\text{sat}}, S_{\text{unSat}}, S_{\text{strategy}}, S_{\text{act}}, S_{\text{aSuccess}}, S_{\text{aFail}}\} \subseteq S$;
4) S_{rSuccess}, S_{rFail}, S_{sat}, S_{unSat}, S_{aSuccess}, and S_{aFail} are end states;
5) $\rightarrow \subseteq S \times S$ is the transition relation, and it is classified into nine types: $\{S_{\text{initial}} \times S_{\text{read}}, S_{\text{read}} \times \{S_{\text{rSuccess}}, S_{\text{rFail}}\}, S_{\text{rSuccess}} \times \{S_{\text{sat}}, S_{\text{unSat}}\}, S_{\text{unSat}} \times S_{\text{strategy}}, S_{\text{strategy}} \times S_{\text{act}}, S_{\text{act}} \times \{S_{\text{aSuccess}}, S_{\text{aFail}}\}\}$;
6) AP is a set of atomic propositions;
7) $L : S \rightarrow 2^{\text{AP}}$ is a labeling function (2^{AP} denotes the power set of AP).

The ML-FSM comprises seven states and seven transitions, and the details regarding the states and transitions are described in Fig. 5 and are given as follows:

1) The initial state (S_{initial}) is the initial state and receives the input value. The initial state reaches each read sensor ($S_{\text{initial}} \times S_{\text{read}}$).
2) The read sensor state (S_{read}) is a set of states related to the slave layer (i.e., sensors); thus, it is responsible for reading and checking data from sensors. Therefore, each sensor state can be replaced with a related SN-FSM to describe the detailed sensor activities. The data from sensors are used to check the requirements, and the status is reached to check the requirement states (i.e., the next transition is decided from the results of the SN-FSM). However, if a sensor successfully returns information, it reaches the read success state (S_{rSuccess}). Conversely, if a sensor returns fail, the status is transferred to the read fail states (S_{rFail}).
3) The read success state (S_{rSuccess}) and read fail state (S_{rFail}) are end states, and these states can verify which sensor is successfully returned information and not. In addition, reaching both states is exclusive because the sensor cannot be successful and fail. The read success state is a set of states that denotes the sensors that return information successfully. The next transition is decided by the results from sensors; thus, if the requirement is satisfied, the status reaches a satisfied state ($S_{\text{rSuccess}} \times S_{\text{sat}}$). In addition, a sensor may exist to collect information that is not related to requirements; thus, the status can be ended at the read success state in this case. In addition, if the requirement is not satisfied, it reaches an unsatisfied state ($S_{\text{rSuccess}} \times S_{\text{unSat}}$). However, the transition from the read success state to the checking requirements state is one-to-many because some sensors may detect multiple factors related to various requirements. However, the read fail state (S_{rFail}) is a set of states that fails to sense the environment; thus, the transition ends.
4) Satisfied state (S_{sat}) and unsatisfied state (S_{unSat}) are end states, and these states can verify which requirements are satisfied and which are unsatisfied. If the status reaches unsatisfied states, the status has to be translated to extract strategies for adaptation ($S_{\text{unSat}} \times S_{\text{strategy}}$). However, if the status reaches satisfied requirement states, adaptation is not required.

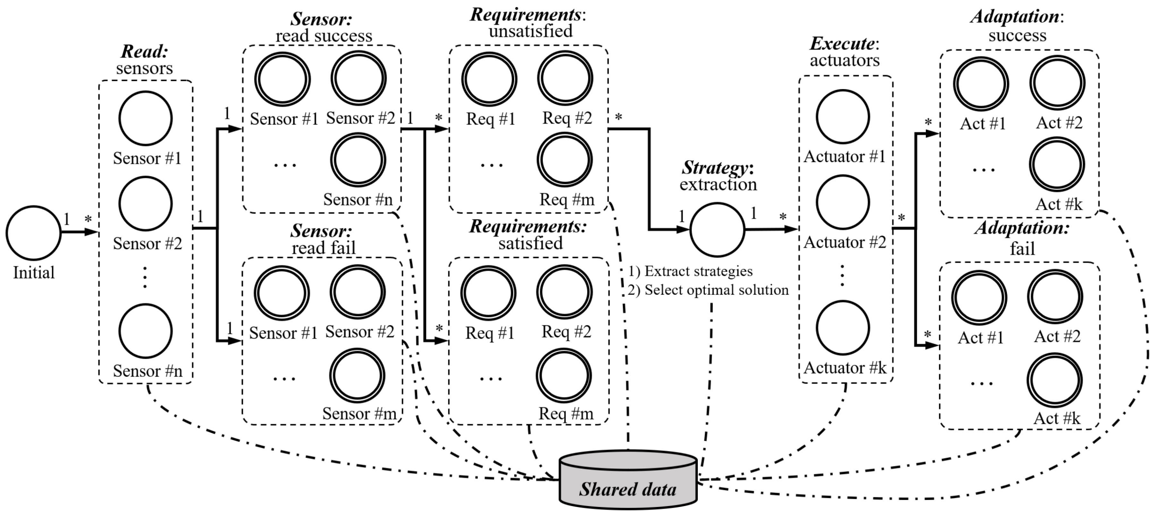


Fig. 5. Finite-state machine to design an overall IoT system.

5) The reaching strategy state (S_{strategy}) denotes that there are unsatisfied requirements; this system has to generate adaptation strategies and find the most optimal solution to accomplish the unsatisfied requirements. The game theory-based strategy extraction method [11] was used to find the strategies and the most optimal solution. However, the most optimal solution contains orders to execute actuators; thus, the status translates into several activate states (S_{act}) to execute actuators.
6) The activation state (S_{sat}) is related to the slave layer (i.e., actuators); thus, it can be replaced with the related ACT-FSM. Each activation state operates the related actuators based on the most optimal solution, and the operation results affect the decision related to the next transition. The status is translated to the success state ($S_{\text{sat}} \times S_{\text{aSuccess}}$) if the operation of the actuator is successfully operated, but it reaches a failed state ($S_{\text{sat}} \times S_{\text{aFail}}$) if the operation fails.
7) Success and fail states (S_{aSuccess}, S_{aFail}) are end states, and these states are related to the verification results of the adaptive strategy execution. If the operation is successfully completed, the success state is reached; if the failure states are reached, the operation fails.

As described previously, the ML-FSM design describes an overall IoT system, and the ML-FSM is designed based on the collected slave devices (i.e., SN-FSM and ACT-FSM). Therefore, if the new slave device is added to the system, the ML-FSM must be updated, but only a few transitions are required. However, the ML-FMS is constructed for an IoT system, and then the designed finite-state machine is abstracted for runtime verification. The details of the abstracted model are described in Section III-B3.

3) Finite-State Machine Design for Verification at Runtime: In this section, a finite-state machine is described that is used to verify the IoT system at runtime. The finite-state machine is an abstracted finite-state machine extracted from the ML-FSM, and the cache-based abstraction method [24] is applied to the abstraction (see Section II-C2). The abstracted master layer finite-state machine, called AML-FSM, is expressed as a tuple (S, $\rightarrow$, S_{initial}, AP, L), where

1) S is a set of states;
2) S_{initial} is an initial state;
3) the states are classified into seven types: $\{S_{\text{initial}}, S_{\text{rSuccess}}, S_{\text{rFail}}, S_{\text{sat}}, S_{\text{unSat}}, S_{\text{aSuccess}}, S_{\text{aFail}}\} \subseteq S$;
4) S_{rSuccess}, S_{rFail}, S_{sat}, S_{unSat}, S_{aSuccess}, and S_{aFail} are end states;
5) $\rightarrow \subseteq S \times S$ is the transition relation, and it is classified into six types $\{S_{\text{initial}} \times \{S_{\text{rSuccess}}, S_{\text{rFail}}, S_{\text{sat}}, S_{\text{unSat}}, S_{\text{aSuccess}}, S_{\text{aFail}}\}$;
6) AP is a set of atomic propositions;
7) $L : S \rightarrow 2^{\text{AP}}$ is a labeling function (2^{AP} denotes the power set of AP).

Fig. 6 shows AML-FSM. As described in the AML-FSM tuple, its states are end states of ML-FSM except for the initial state; thus, the definitions of the end states are the same as those for ML-FSM (i.e., the details of the states of AML-FSM are skipped because they are the same as those of the states of ML-FSM). The purpose of the abstraction process (i.e., the abstracting process in Fig. 2) is to find reachable paths from the initial state to the end states of the ML-FSM. Therefore, each transition in AML-FSM connects the initial states and each end state, and the transitions are composed of simple logical equations or mathematical equations. Therefore, the status of AML-FSM can be determined by only calculating the equations, which significantly reduces the verification time. However, the AML-FSM is used to verify the IoT system at runtime, and the details of the runtime verification are described in Section III-C.

C. Verification With Model Checking at Runtime

In this section, the runtime verification process is described in detail (i.e., the managing processes in the master layer). The verification process consists of a MAPE-K loop, which is generally applied in the self-adaptive software [50]. The details of each step are as follows.

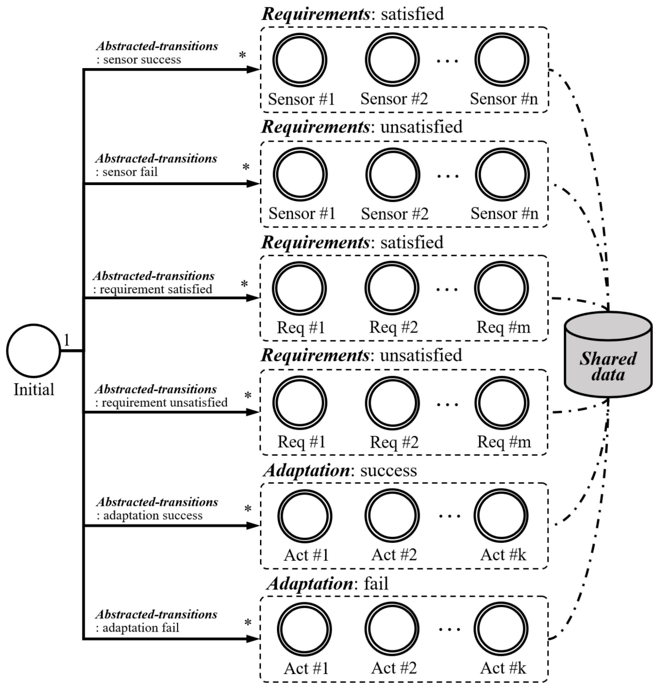

Requirements: satisfied
Abstracted-transitions : sensor success
Sensor #1
Sensor #2
Sensor #n
Requirements: unsatisfied
Abstracted-transitions : sensor fail
Requirements: satisfied
Abstracted-transitions : requirement satisfied
Req #1
Req #2
Req #m
Requirements: unsatisfied
Abstracted-transitions : requirement unsatisfied
Adaptation: success
Abstracted-transitions : adaptation success
Act #1
Act #2
Act #k
Adaptation: fail
Abstracted-transitions : adaptation fail
Initial
1
*
Shared data

Fig. 6. Finite-state machine for verification at runtime.

1) Monitoring Process: The monitoring process is the first step in the verification process. The monitoring process is responsible for checking and updating data from the slave layer (i.e., sensors and actuators) and the master layer (i.e., the IoT system). The collected data are saved in the shared database (i.e., shared knowledge in MAPE-K). In addition, if a new slave device is detected, the system model has to be updated; thus, the monitoring process performs the modeling phase (i.e., the new device is added to the previous ML-FSM, and the ML-FSM is abstracted). After the monitoring process, the analysis process is performed.

2) Analysis Process: The analysis process uses an abstracted finite-state machine form (AML-FSM) to verify the system's status. As described previously, the reason for using AML-FSM for verification at runtime is that AML-FSM transitions are simplified equations; thus, the reachable paths can be calculated quickly at runtime. However, performance experiments to demonstrate the effectiveness of the proposed approach are described in Section IV.

As described in Section III-B3, AML-FSM defines reachability from the initial state to the states related to requirements (i.e., satisfied and unsatisfied) and adaptation (i.e., success and fail). Therefore, the results of requirement-related states can verify which requirements are satisfied or unsatisfied. If the status reaches a satisfied state, a requirement related to the state is satisfied. In addition, if the status reaches an unsatisfactory state, a requirement related to the state is unsatisfied. A requirement cannot be both conditions; thus, the reachability to the satisfied and unsatisfied states is exclusive for the same requirement. Similarly, the reachability of adaptation-related states verifies that the execution of adaptation strategies is successfully performed. The reaching success state (S_{success}) denotes that an adaptation strategy is successfully performed, but if it reaches the fail state (S_{fail}), an adaptation strategy failed to execute. In addition, both end states are exclusive. However, reaching the success state guarantees that an adaptation is successfully performed, but it is not guaranteed that an adaptation strategy changes the unsatisfied requirements to satisfy the conditions. Therefore, the master layer (i.e., the host of the IoT system) must continually verify the requirement conditions.

With verification results from AML-FSM, the system can analyze 1) which requirements are required to adapt and 2) which adaptation strategies are successfully operated. The analyzed results are updated in the shared database. After the analysis process, the planning process was executed.

3) Planning Process: The proposed approach applies a game theory-based strategy extraction method [11], [12]. The method uses the Nash equilibrium, which is a decision-making theorem within the game theory, and the strategy extraction method showed a reasonable amount of time with complex IoT environments. The details of the strategy extraction method are beyond the scope of this article; thus, the method is briefly described with definitions of game theory. In the game theory, players can select several strategies that can affect other players' choices. Therefore, a payoff matrix is required to check the influences between players' strategies, and the optimal strategy is extracted based on the payoff matrix. The goal of the optimal strategy is to achieve as many players' satisfaction as possible. In addition, some rules are required to determine the optimal solution using the payoff matrix.

In the proposed approach, the analysis process extracts candidate strategies using a game-theory-based method and data from the shared database. A situation that requires adaptation is defined as a game. Satisfied requirements are not required to be adapted; thus, unsatisfied requirements are players of a game. In addition, possible actions that may be a solution to unsatisfied requirements are the strategies of the unsatisfied requirements (i.e., possible executions of actuators related to the unsatisfied requirements are set as strategies of the player). Then, the analysis process generates a payoff matrix and extracts candidate strategies using data from the shared database.

For example, there are two requirements for sensors: 1) light intensity and 2) humidity. Both requirements are not satisfied: the requirements are lower than the satisfaction criteria, and the outdoors have brighter light intensity and higher humidity than indoors. The actuators are a lamp and windows. The lamp can affect the light requirement by adjusting the brightness, and the windows operation can affect both requirements. The game is to determine the most appropriate strategy for both requirements, and players are the requirements. In addition, the possible action of the actuators can be solutions, and the results are denoted using the payoff. The results of the game (i.e., the operation of the actuators) are "may be satisfied by operating actuators" and "not affected by operating actuators," and payoffs are 1 and 0, respectively. In this situation, the payoff table is generated, as shown in Table II, and the results denote the payoffs of the requirements. The former is the payoff of the light requirement, and the latter is the payoff of the humidity requirement. In the table, there are two candidate strategies

TABLE II
EXAMPLE OF PAYOFF MATRIX

		Lamp	
		On	Off
Windows	Open	**(1,1)**	**(1,1)**
	Close	(1,0)	(0,0)

that can perform adaptation: 1) opening windows and turning on the lamp and 2) only opening the windows. In addition, both candidate strategies have the possibility to satisfy the requirement; thus, they are in Nash equilibrium. Therefore, to determine the optimal solution, an evaluation method is required to evaluate the candidate strategies (Table II).

An equation to determine the most optimal solution is applied [11]. The equation consists of three major conditions to evaluate the strategies.

1) The number of satisfied requirements (SR) is the number of requirements that can be satisfied by executing a strategy. A strategy that satisfies multiple requirements is preferable.
2) The number of related requirements (RR) is the number of requirements that can be affected by strategy execution. A strategy that affects fewer related requirements is preferable. For example, if windows are opened to adapt to brightness, other requirements (i.e., humidity, temperature, and dust density) may be affected.
3) The number of actuators (AD) is the number of devices that operate under an adaptive strategy. Smaller values are preferable for AD. For example, if there are several strategies, and they may adapt the same requirements, then the strategy resulting in the operation of fewer ADs is preferable.

The equation used to evaluate adaptive strategies using SR, RR, and AD is described as follows:

$$\alpha\left\{\log\left(\frac{\mathrm{SR}+1}{\mathrm{RR}+1}+1\right)\right\}+\beta\left\{\log\left(\frac{1}{\mathrm{AD}+1}+1\right)\right\}. \quad (1)$$

The first term (i.e., $\{\log([(\mathrm{SR}+1)/(\mathrm{RR}+1)]+1)\}$) is related to requirements; thus, requirement-related factors (i.e., SR and RR) are used. The second factor (i.e., $\{\log([1/(\mathrm{AD}+1)]+1))$ is related to the operation of actuators; thus, an actuator-related factor (i.e., AD) is applied. Both terms use a logarithm for normalization, and 1 is added to prevent an infinite result. The coefficients (α and β) give weights of two terms, and their summation is 1. In the example scenario, there are two candidate strategies, and each SS is calculated as 0.41 (i.e., SS of opening windows and turning on the lamp) and 0.5 (i.e., SS of only opening the windows). Note that α and β are set to 0.5. Finally, the optimal solution of the example is extracted as "only opening windows." Both strategies may satisfy the requirements, but the optimal solution operates by running fewer actuators.

With candidate strategies extracted using the game theory-based method, the planning process selects the most optimal solution (i.e., the highest score strategy from the candidate strategy). In addition, the data generated from the planning process are saved in a shared database, and the optimal solution is transferred to the execution process.

4) Executing Process: The executing process is responsible for operating the actuators for adaptation. The operation is performed based on the results of the planning process; thus, if there is a strategy to adapt, actuators are executed as described in the adaptive strategy. However, if there is no adaptation strategy (i.e., all requirements are satisfied, or an adaptation strategy is not extracted), the execution process maintains the current status. The execution process is the end of the MAPE Loop; thus, the loop continues after the execution process.

IV. EMPIRICAL EVALUATION

The proposed method was implemented as a prototype to evaluate the model abstraction and verification performance. The prototype was implemented using Java 15.0.4, and the test device was compromised with Intel Core i7-10700K (3.80 GHz and 8 cores), 32-GB memory, and Windows 10. Experimental data were randomly generated, and the details are described in Section IV-A. The results of the experiments for model abstraction and runtime verification are described in Sections IV-B and IV-C, respectively.

A. Experimental Data Set

To perform the experiment, experimental data were designed to generate different IoT environments with a number of actuators, requirements, and complexity. The experimental IoT environment is compromised by four complexity factors, and the details of the factors are described as follows.

1) The number of actuators (NA) is related to the complexity of the actuator size. An IoT environment with larger actuators is more complex than that with small actuators if the other options are the same.
2) The number of requirements (NR) is related to complexity with the requirement size. Similar to NA, it is assumed that more requirements make a more complex IoT environment if the other conditions are constant.
3) The additional affected requirement (AR) denotes the requirements that are affected by the operation of single actuators; for example, if an actuator is related to three requirements (i.e., AR is 3), then the operation of the actuator may affect the three related requirements. In addition, increasing the AR can create more complex IoT environments. However, as a default, an actuator can affect only one requirement.
4) The proportion of actuators (PA) denotes the percentage of actuators that have AR; thus, if PA is increased, then the IoT environment is more complex.

Fig. 7 denotes an abstracted model to describe the relationship between the complexity factors. In the figure, there are *n* actuators and *k* requirements; therefore, NA and NR are *n* and *k*, respectively. In addition, *m* actuators are included in PA (i.e., PA is equal to *m*/*n*), and actuators included in PA are associated with AR requirements. As a default, the remaining actuators (i.e., *n*-*m* actuators) are connected to requirements with one-to-one relationships. Based on the complexity factors, we generated three experimental environments, and each data set was related to single complexity factors. The details are described as follows.

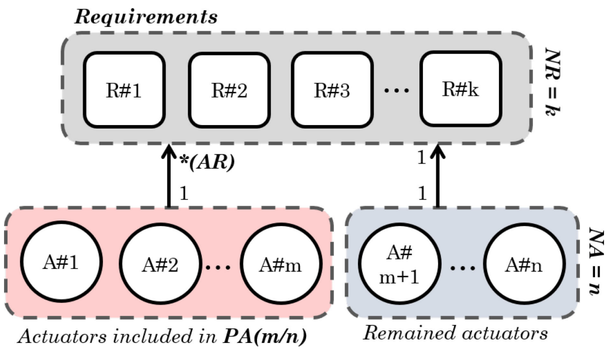


Fig. 7. Relationship between experimental factors for IoT complexity.

1) Experimental environment #1 (*EE#1*) is related to increasing NA (i.e., 20–50); thus, other factors are fixed (i.e., NR, AR, and PA are set as 10, 2, and 10%).
2) Experimental environment #2 (*EE#2*) is related to increasing NR (i.e., 2–50); thus, other factors are fixed (i.e., NA, AR, and PA are set as 50, 2, and 10%).
3) Experimental environment #3 (*EE#3*) is related to increasing PA (i.e., 0%–30%); thus, other factors are fixed (i.e., NA, NR, and AR are set as 50, 10, and 2).

Furthermore, we generated 100 environments with the same factor values, but the connections between the requirements and actuators were generated well randomly. Therefore, each experimental IoT environment is different from the others (i.e., there are 100 different IoT environments with the same complexity factor values). In addition, all requirements have at least one connected actuator; thus, there is no requirement that is not affected by the actuators. After the experimental data setting, we performed model abstraction and verification experiments, and the details and results are described in the following section.

B. Experiment for Abstraction Performance

The experiments were performed to evaluate the model abstraction and runtime performance. To evaluate the abstraction performance, the generated IoT environments were transferred to the model for the proposed approach and RINGA-IoT [12], and then the abstraction processes were executed. Figs. 8–10 show the results of the comparison between the proposed approach and RINGA-IoT. Fig. 8 shows the results obtained for the first experimental environment with increasing number of actuators. Naturally, more actuators require more time for model abstraction (i.e., RINGA-IoT has time from 2.8 to 111.4 ms, and the proposed method has time from 0.09 to 0.16 ms), but the cache-based method [24] is applied in the proposed approach, which significantly reduces the abstraction time. In addition, the proposed approach requires a reasonable time for model abstraction.

The model abstraction results with *EE#2* are shown in Fig. 9. In contrast to the previous results, the model abstraction from RINGA-IoT shows a tendency to decrease after rising. This is because the model from RINGA-IoT is highly complex when requirements and actuators are connected in a complex manner. In addition, in the experimental environment,

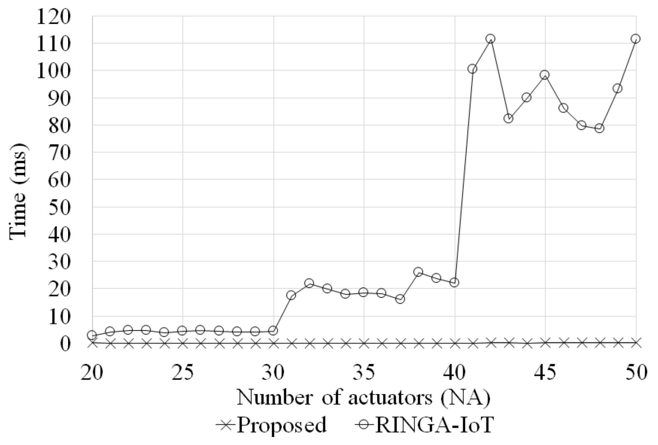


Fig. 8. Results of model abstraction time with increasing actuators (NA).

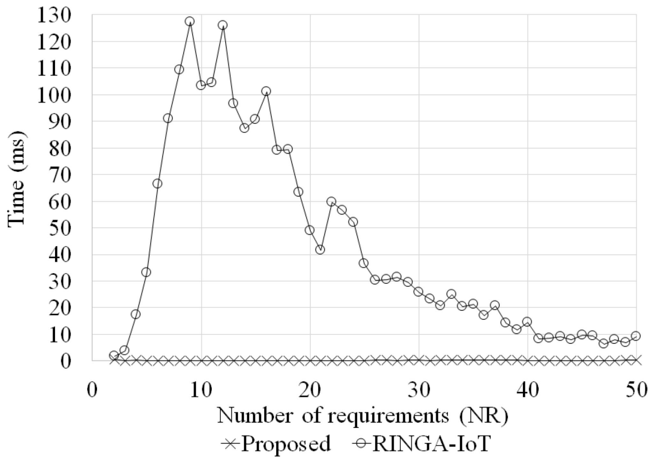


Fig. 9. Results of model abstraction time with increasing requirements (NR).

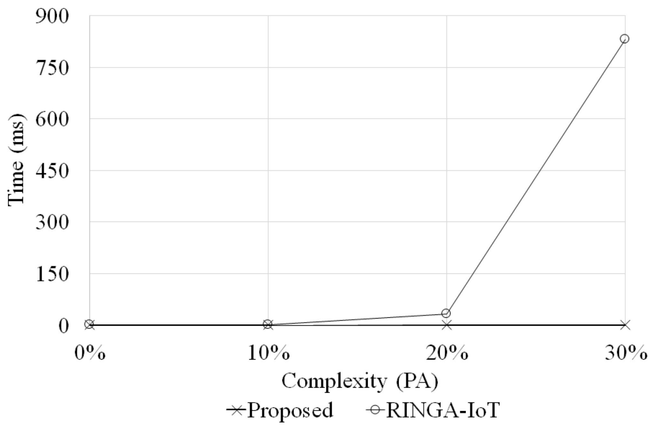


Fig. 10. Results of model abstraction time with complexity (PA).

the number of actuators is fixed (i.e., 50); thus, the model complexity decreases after increasing. The abstraction results from the proposed approach are steadily increased from 0.23 to 0.29 ms because more cached data are required when the requirements are increased. However, the results also showed that the proposed approach has a more reasonable model abstraction performance than the previous abstraction method.

Fig. 10 shows the results with increasing PA factor. As described, the complexity factor is related to the percentages of actuators that are related to many requirements; thus, the results show that both abstraction methods require more abstraction time when more complexity is required. The time required for RINGA-IoT was 0.28–329.9 ms, and the required time for the proposed approach was 0.16–0.21 ms. The results show that the proposed approach is more practicable than RINGA-IoT for the model-abstraction process. In short, the overall results show that the proposed approach significantly reduces the model abstraction performance because the cache-based mechanism is applied, which helps reduce the abstraction time.

C. Experiment for Runtime Verification Performance

In this section, experiments related to runtime verification are described. For the experiments, the abstracted models that were generated in the previous experiment (i.e., Section IV-A) were used because the proposed approach and RINGA-IoT required abstraction models for verification at runtime. In addition, for comparison with other model-checking tools (i.e., cadenceSMV [25], NuSMV [26], [27], and nuXmv [28], [29]), the experimental IoT environments were transferred to other model-checking languages. The other tools use the same FSM model with the proposed approach; thus, performances to find reachability from initial states to three types of end states [i.e., *n* satisfies requirements (*SR*), *m* dissatisfies requirement (*DR*), and *k* adaptable strategy (*AS*)] are measured. The reachability is described using the intuitive semantics of temporal modalities [30] as follows:

$$\delta = \bigwedge_{i=1}^{n} \exists\Diamond \mathrm{SR}_i + \bigwedge_{i=1}^{m} \exists\Diamond \mathrm{DR}_i + \bigwedge_{i=1}^{k} \exists\Diamond \mathrm{AS}_i.$$

Symbols "$\exists$" and "$\Diamond$" denote "exists" and "eventually," respectively. In addition, "$\bigwedge$" indicates conjunction. Therefore, "$\bigwedge_{i=1}^{n} \exists\Diamond SR_i$" will be true if all requirements are satisfied, and all requirements are verified to find an answer "$\bigwedge_{i=1}^{n} \exists\Diamond SR_i$" Finally, δ is required to check all requirements and adaptable strategies, similar to the proposed approach.

Figs. 11–13 show the results of runtime verification. Fig. 11 shows the results for the first IoT experimental environment (i.e., *EE*#1) with increasing actuators. Complete tools gradually require more computational time for a more complex model (the proposed approach requires a time from 0.09 to 0.16 ms), but RINGA-IoT increases rapidly when the IoT environments become complicated. However, the figure shows that the proposed approach requires the least computation time to verify the model.

The runtime experiment results with increasing requirements are shown in Fig. 12. Except RINGA-IoT, the other tools, including the proposed approach, required more time when the requirements were increased. RINGA-IoT showed a tendency to decrease after rising. The reason is the same as that for the model-abstraction process, that is, the RINGA-IoT model generates complex models (i.e., RINGA-IoT requires more time to verify the complex model) when the actuators affect several requirements; thus, it has a high computing time with the NRs between 10 and 20. However, the proposed approach required more time for verification in fields (i.e., requirement number is larger than 38), but the proposed approach is more effective than RINGA-IoT from the overall perspective, including the model-abstraction process. In addition, as depicted in Fig. 12, the proposed approach shows better performance than the other model-checking tools overall.

The results related to PA complexity are depicted in Fig. 13. Naturally, the proposed approach required more time when the

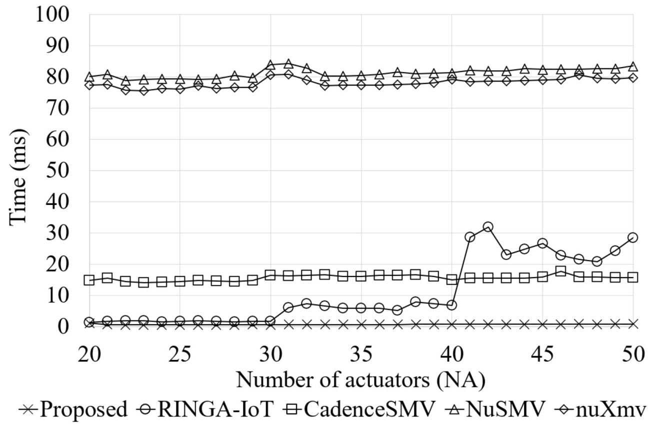


Fig. 11. Results of verification time with increasing actuators (NA).

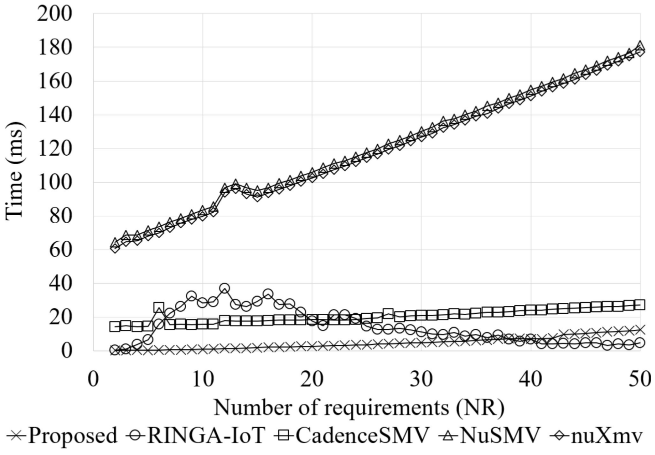


Fig. 12. Results of verification time with increasing requirements (NR).

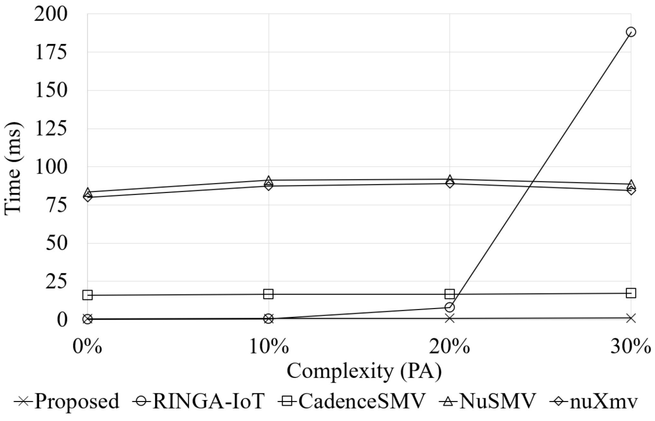


Fig. 13. Results of verification time with complexity (PA).

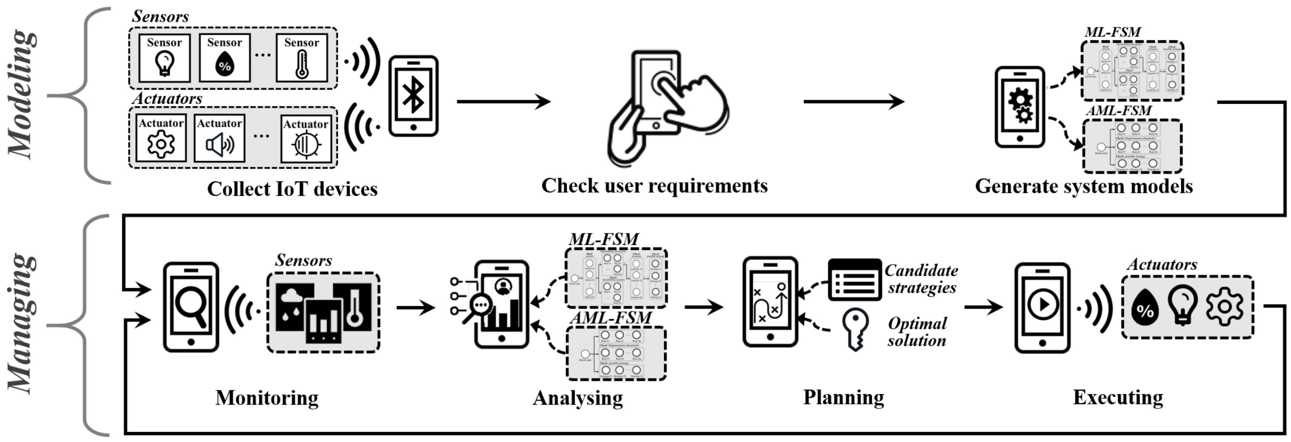


Fig. 14. Overview of the example application.

complexity was high (i.e., the proposed approach required time from 0.7 to 1.07 ms), but the other tools provided monotonic results. This is because the proposed approach and RIGNA-IoT consider all possible paths to find reachability, but the other tools terminate to find a reachable path if they find a single reachable path. Therefore, if the PA complexity does not affect the model-checking time in NuSMV, nuXmv, and CadenceSMV. Nevertheless, the proposed approach is superior to the verification of other tools.

In summary, the proposed approach shows reasonable verification results at runtime, and the overall results are better than those of other model-checking tools. In addition, the cache-based mechanism significantly reduced not only the abstraction process but also the runtime verification time. In addition, the experimental data set was generated as different IoT environments with the four complexity factors; thus, the data set was generated from a simple IoT environment to complex IoT environments. Furthermore, the proposed approach shows a stable verification time; thus, the proposed approach can be applied to complex IoT environments with a stable verification performance.

V. Proof of Concept: IoT-Based Smart Greenhouse

In this section, a small-scale IoT-based smart greenhouse application is introduced to help understand the proposed approach, and the application demonstrates the efficacy of the proposed framework. The application comprises four scenarios to verify and adapt to various situations.

A. Overview of IoT-Based Smart Greenhouse

In the example application, requirements and IoT devices are collected during the modeling phase. There are two types of requirements for cultivation: 1) rapidly changing requirements (i.e., intensity of brightness) and 2) gradually changing requirements (i.e., humidity). The former can rapidly change when the operation of a related device immediately affects the environment. For example, the intensity of brightness immediately changes when the lamp is turned on or off. In contrast, in the latter, the operation of a related device results in gradual changes. For example, humidity can be controlled by operating a humidifier; however, it can only be changed gradually. In addition, the example application involves several IoT actuators (i.e., lamps, humidifiers, and windows and sensors (i.e., illumination and humidity sensors). The lamp and humidifier are actuators, and their operations directly affect environmental changes. The former can adjust brightness, and the latter can increase humidity. In addition, windows can adjust brightness and humidity by opening and closing them, but their effects depend on the outside environment. Therefore, the window directly changes the environment. Sensors sense the intensity of brightness and humidity and are located indoors and outdoors to collect information from different areas. All slave devices are equipped with wireless modules; thus, it is possible to communicate with a host device (i.e., master layer) that collects information on IoT devices located in its network range. In addition, a smartphone (i.e., Samsung Galaxy S8, 2.31-GHz CPU clock with 8 core and 8-GB memory) is a host device in this application. Subsequently, a user can set the requirements based on the collected IoT devices (i.e., slave devices), and the host device constructs an ML-FSM and an AML-FSM to adapt to the environment at runtime.

After the modeling phase, the managing phase is executed following the MAPE loop. The host device collects data from slave devices, such as indoor and outdoor humidity, available devices, and requirements. After monitoring, the host analyzes monitored data using AML-FSM, and then the planning process extracts adaptive candidate strategies using the game theory-based method and selects the most optimal solution using the strategy score [11]. The optimal solution is executed in the execution process, and the loop continues. However, it is assumed that the slave devices provide information regarding related requirements and influences to requirements. Therefore, the host device can generate a system model that is automatically constructed using the definition of ML-FSM that is used for verification at runtime. Fig. 14 shows an overview of the application of this example.

In an example application, both requirements should be satisfied under various environmental changes. Thus, four scenarios were developed to assess the adaptability of the proposed example application.

1) *Scenario #1:* The humidity requirement is satisfied during the experiment, and the application is adapted to

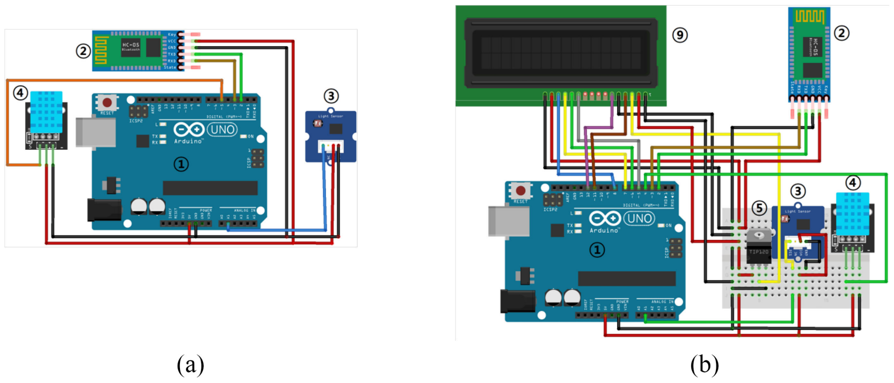


Fig. 15. Components and deployment of sensor devices. (a) Outdoor sensor device. (b) Indoor sensor device.

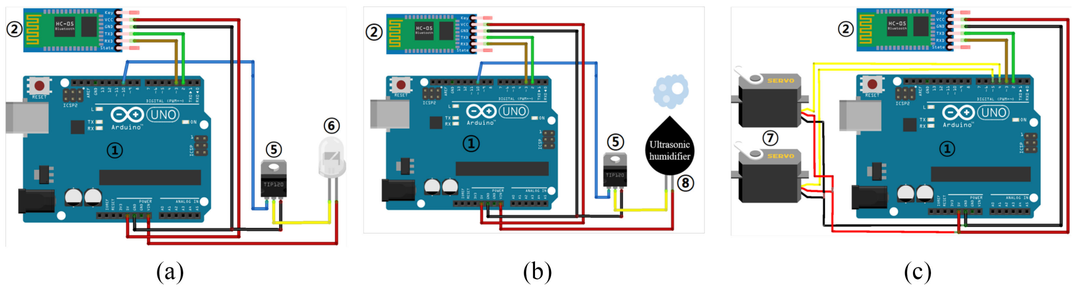


Fig. 16. Components and deployment of actuators. (a) Lamp. (b) Humidifier. (c) Windows.

satisfy the light requirement under dynamic external changes.

2) *Scenario #2:* The light requirement is satisfied during the experiment, and the application is adapted to satisfy the humidity requirement under stable external changes.
3) *Scenario #3:* Neither requirement is satisfied, but there are strategies to satisfy both requirements.
4) *Scenario #4:* The light requirement is satisfied, but the humidity requirement is not. A strategy is extracted, but it causes environmental changes. Therefore, a strategy should be extracted to adapt to light requirements.

Based on the example application and scenarios, the application and IoT devices are implemented in Sections V-B1 and V-B2. The results of these scenarios are described in Section V-C.

B. Implementation

1) Arduino Implementation: The slave devices were implemented using an Arduino. Table III lists the hardware components required to implement the sensors and actuators. Fig. 15 shows the deployment of the sensor components. The Arduino UNO board [61] is an open-source microcontroller board based on the ATmega328P microcontroller. The Bluetooth module HC-05 was connected to a wireless network. In addition, an illumination sensor and a humidity/temperature sensor DHT 11 were used. The sensor devices were located indoors and outdoors. Fig. 15 shows the components of the outdoor sensor device. The indoor sensors [Fig. 15(b)] were similar to the outdoor sensor device, except for an LCD panel that was used to show the status of the requirements.

TABLE III
HARDWARE COMPONENT LIST

Number	Component	Description
1	Arduino UNO [61]	Microcontroller board module
2	HC-05	Bluetooth SPP (Serial Port Protocol) module
3	Illumination sensor	Senses the intensity of illumination
4	DHT 11	Senses humidity and temperature
5	Transistor	Amplifies electrical power to adjust connected device
6	LED strips	Adjusts brightness
7	Servo motor	Opens and closes window
8	Humidifier	Increases humidity using ultrasonic humidifier
9	LCD module	Shows lamp status

Fig. 16 shows the deployment of actuator components. Similar to the sensor devices, the actuators were based on the Arduino UNO board, and the HC-05 module was used. The actuators were designed with an operating range of 0–255. The minimum value of 0 implies that the device is turned off, whereas the maximum value of 255 implies that the device operates at full capacity. The actuators have different operating ranges, as described below. LED strips consisting of LED lights were used to adjust the illumination, and the LED strips were amplified using a transistor. Fig. 16(a) shows the component deployment of the actuator for controlling lighting. The operation range of the lamp was 5; thus, the lamp had 51 operation levels (i.e., 0, 5, 10, 15, …, 255). An ultrasonic humidifier was used in the actuator for humidity control. The ultrasonic humidifier was rated at 2 W, dc 5 V, and 350 mA. Thus, the Arduino board was able to power the humidifier. A

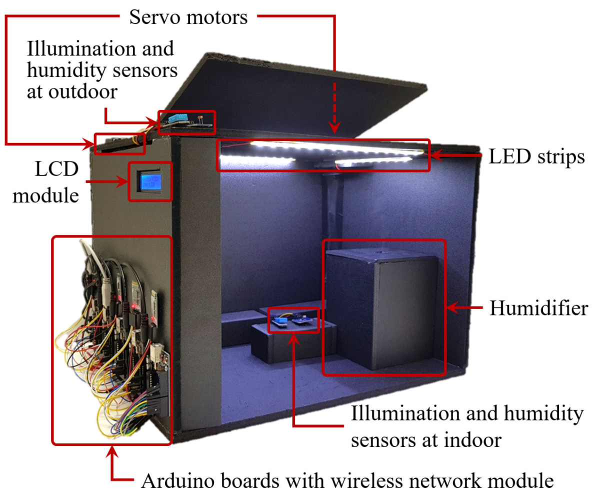
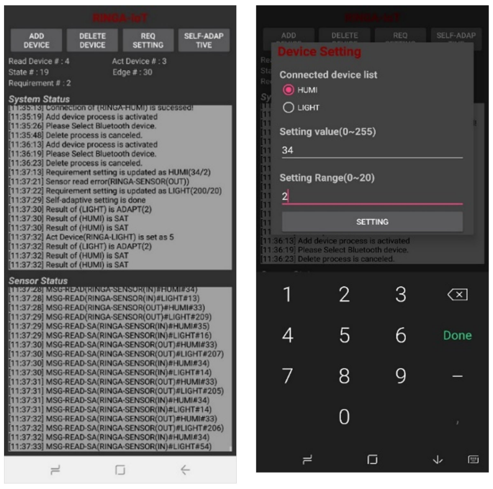

Servo motors
Illumination and humidity sensors at outdoor
LED strips
LCD module
Humidifier
Illumination and humidity sensors at indoor
Arduino boards with wireless network module

Fig. 17. Experimental environment with sensors and actuators.



```
Read Device # : 4          Act Device # : 3
State # : 19               Edge # : 30
Requirement # : 2
System Status
[11:35:13] Connection of (RINGA-HUMI) is sucessed!
[11:35:19] Add device process is activated
[11:35:26] Please Select Bluetooth device.
[11:35:48] Delete process is canceled.
[11:36:13] Add device process is activated
[11:36:19] Please Select Bluetooth device.
[11:36:23] Delete process is canceled.
[11:37:13] Requirement setting is updated as HUMI(34/2)
[11:37:21] Sensor read error(RINGA-SENSOR(OUT))
[11:37:22] Requirement setting is updated as LIGHT(200/20)
[11:37:29] Self-adaptive setting is done
[11:37:30] Result of (LIGHT) is ADAPT(2)
[11:37:30] Result of (HUMI) is SAT
[11:37:30] Result of (HUMI) is SAT
[11:37:32] Act Device(RINGA-LIGHT) is set as 5
[11:37:32] Result of (LIGHT) is ADAPT(2)
[11:37:32] Result of (HUMI) is SAT
[11:37:32] Result of (HUMI) is SAT
Sensor Status
[11:37:28] MSG-READ(RINGA-SENSOR(IN)#HUMI#34)
[11:37:28] MSG-READ(RINGA-SENSOR(IN)#LIGHT#13)
[11:37:28] MSG-READ(RINGA-SENSOR(OUT)#HUMI#33)
[11:37:29] MSG-READ(RINGA-SENSOR(OUT)#LIGHT#209)
[11:37:29] MSG-READ-SA(RINGA-SENSOR(IN)#HUMI#35)
[11:37:29] MSG-READ-SA(RINGA-SENSOR(IN)#LIGHT#16)
[11:37:30] MSG-READ-SA(RINGA-SENSOR(OUT)#HUMI#33)
[11:37:30] MSG-READ-SA(RINGA-SENSOR(OUT)#LIGHT#207)
[11:37:30] MSG-READ-SA(RINGA-SENSOR(IN)#HUMI#34)
[11:37:30] MSG-READ-SA(RINGA-SENSOR(IN)#LIGHT#14)
[11:37:31] MSG-READ-SA(RINGA-SENSOR(OUT)#HUMI#33)
[11:37:31] MSG-READ-SA(RINGA-SENSOR(OUT)#LIGHT#205)
[11:37:31] MSG-READ-SA(RINGA-SENSOR(IN)#HUMI#34)
[11:37:31] MSG-READ-SA(RINGA-SENSOR(IN)#LIGHT#14)
[11:37:32] MSG-READ-SA(RINGA-SENSOR(OUT)#HUMI#33)
[11:37:32] MSG-READ-SA(RINGA-SENSOR(OUT)#LIGHT#206)
[11:37:32] MSG-READ-SA(RINGA-SENSOR(IN)#HUMI#34)
[11:37:33] MSG-READ-SA(RINGA-SENSOR(IN)#LIGHT#54)
```

(a)

```
Device Setting
Connected device list
(●) HUMI
( ) LIGHT
Setting value(0~255)
34
Setting Range(0~20)
2
[11:36:13] Add device process is activated
[11:36:19] Please Select Bluetooth device.
[11:36:23] Delete process is canceled.
```

(b)

Fig. 18. Screenshots of the implemented Android application. (a) Runtime. (b) Setting requirement.

transistor was also used to amplify the humidifier. Fig. 16(b) shows the component deployment of the actuator for humidity control. The humidifier can be only on or off; thus, its operation range is 255 (i.e., the humidifier has an operation value of only 0 or 255), and it has two levels of operation (i.e., 0 and 255). Two servo motors were used to control the window. They lifted the window up and down. Fig. 16(c) shows the component deployment of the window control device. The operation range of the window was 45; thus, it had six operation levels (i.e., 0, 45, 90, 135, 180, and 255).

A darkroom was constructed for the IoT environment experiments using the sensors and the actuator deployments (Fig. 17).

2) Android Implementation: An Android application was implemented in the experiment. Fig. 18 shows the screenshots of the implemented application, where buttons for various operations are at the top of the screen. Users can manually add or delete IoT devices by touching. They can set the threshold of a requirement by the "REQ SETTING" button [Fig. 18(b)]. Requirements can be selected from the requirement list, and the standard threshold and its range are set as integer numbers. Touching the "SELF-ADAPTIVE" button runs the self-adaptive loop. In addition, various types of information appear under the button (i.e., number of connected sensor, actuators, states, transitions, and requirements). The textbox "System status" displays log data for the application status, and "Sensor status" displays the log data for the connected devices.

C. Implementation

A simulation with the Android application and Arduino IoT devices was conducted based on the scenarios defined in the previous sections. As mentioned earlier, the example application collects IoT devices and constructs the ML-FSM model in the early stage. Fig. 19 shows the ML-FSM modeling results, and Fig. 20 shows the AML-FSM modeling results. The ML-FSM consists of 57 states and 70 transitions, and the states and transitions assign identity numbers (i.e., s0–s56 for states, and e0–e69 for transitions). Note that the identification numbers are not described in Fig. 19 to avoid complexity. In addition, the details of the slave device-related states (i.e., sensors and actuators) and transitions are abstracted as single states. The slave device-related states were designed based on definitions described in Section III-B1. After the design of the ML-FSM, ML-FSM is abstracted using a cache-based method [24]. Abstraction is performed to extract paths from the initial state to the end states (i.e., SsSuccess, SsFail, Ssat, SunSat, Ssuccess, and Sfail). In the abstraction process, the cache-based equations are saved in the shared database, and the data are used for verification at runtime.

Figs. 21–24 show the results for each scenario. The results consist of three graphs: 1) sensed light values; 2) sensed humidity values; and 3) actuator status. The first graph shows the sensed light intensity collected by the indoor and outdoor sensors. The second graph shows the sensed humidity collected by indoor and outdoor sensors. The green area in the first and the second graphs indicate a satisfiable range of requirements; thus, if the sensed value lies within the green area, the requirement is satisfied, whereas if the sensed value is outside the green area, the requirement is dissatisfied. Thus, adaptive strategies are extracted when the sensed value is outside the green area. The red areas indicate that the user requirement is not satisfied, and thus, the system should adapt to environmental changes. In each red area, a box containing related adaptation strategies and requirement status exists. Table IV lists the extracted strategies and the requirement statuses for the example application. The last graph shows the changes in the operation of the actuators using an adaptive strategy. As mentioned in the previous section, actuators have operation values between 0 and 255; however, the operational ranges and levels are different.

Fig. 21 shows the results for Scenario #1. Under this scenario, adaptation is required only for the light requirement (i.e., humidity is stable) owing to sudden changes in the external brightness. As seen in Fig. 21, the application adapts between 0 and 4.5 s, and there are two strategies at that time (strategy #1 in Table IV). The application selects a strategy that increases the lamp brightness and then remains stable until

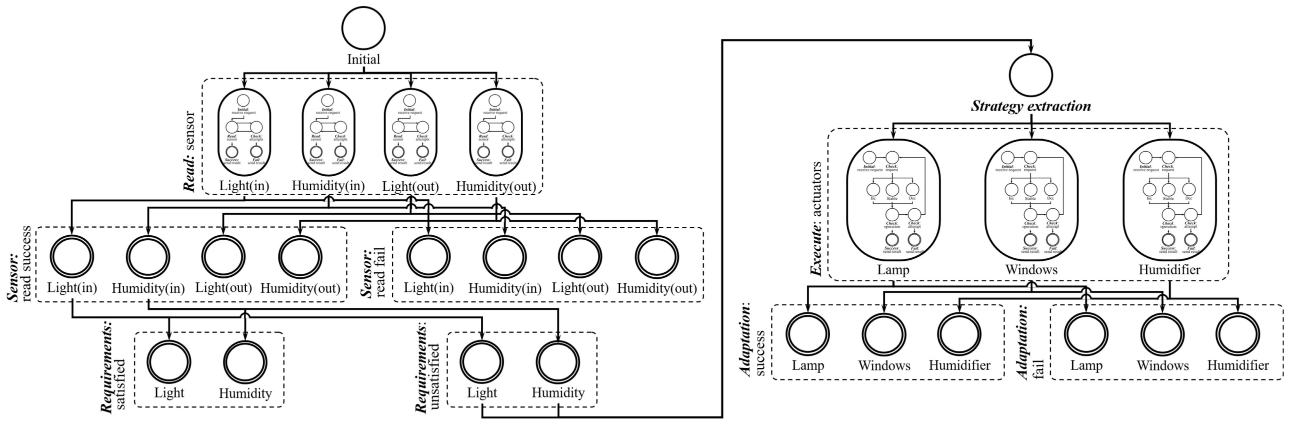
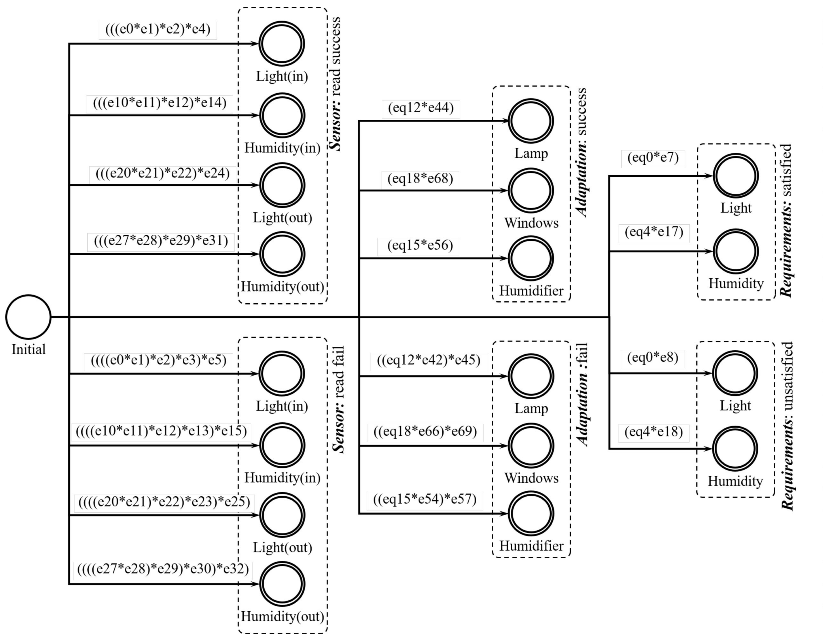


Fig. 19. ML-FSM for example application.



Cached-data in shared database

Equation id	Details equation
eq0	(((e0*e1)*e2)*e4)
eq1	((((e0*e1)*e2)*e3)*e5)
eq2	(eq0*e7)
eq3	(eq0*e8)
eq4	(((e10*e11)*e12)*e14)
eq5	((((e10*e11)*e12)*e13)*e15)
eq6	(eq4*e17)
eq7	(eq4*e18)
eq8	(((e20*e21)*e22)*e24)
eq9	((((e20*e21)*e22)*e23)*e25)
eq10	(((e27*e28)*e29)*e31)
eq11	((((e27*e28)*e29)*e30)*e32)
eq12	((((((((eq3*e9)+(eq7*e19))*e34)*e35)*e36)*e39) +((((((eq3*e9)+(eq7*e19))*e34)*e35)*e37)*e40)) +((((((eq3*e9)+(eq7*e19))*e34)*e35)*e38)*e41))
eq13	(eq12*e44)
eq14	((eq12*e42)*e45)
eq15	((((((((eq3*e9)+(eq7*e19))*e46)*e47)*e48)*e51) +((((((eq3*e9)+(eq7*e19))*e46)*e47)*e49)*e52)) +((((((eq3*e9)+(eq7*e19))*e46)*e47)*e50)*e53))
eq16	(eq15*e56)
eq17	((eq15*e54)*e57)
eq18	((((((((eq3*e9)+(eq7*e19))*e58)*e59)*e60)*e63) +((((((eq3*e9)+(eq7*e19))*e58)*e59)*e61)*e64)) +((((((eq3*e9)+(eq7*e19))*e58)*e59)*e62)*e65))
eq19	(eq18*e68)
eq20	((eq18*e66)*e69)

Fig. 20. Abstracted ML-FSM for example application.

24 s. The illumination of the environment suddenly increases from 103 to 194 lx at 24 s. Initially, the application selects strategy #4 and decreases lamp brightness. However, the application does not adapt to environmental changes, and the lamp brightness remains high until 30 s. Therefore, the application selects a candidate strategy #4 (i.e., strategy #10) and closes the window. The strategy is effective in reducing indoor lighting; however, the reduction becomes excessive at 34.5 s. Therefore, the application extracts strategy #2 and increases lamp brightness until 40.5 s. The light requirement is maintained at 42 s. The illumination of the environment suddenly decreases from 192 to 95 lx at 73.5 s; however, the application requires no adaptation because the light requirement has already been satisfied.

Fig. 22 shows the results for Scenario #2. Under this scenario, adaptation is required only for the humidity requirement (i.e., light is stable). Initially, the humidity requirement should increase; thus, the application turns on the humidifier at 0 s (i.e., strategy #7). However, humidity is a gradually changing requirement, and adaptation proceeds gradually as the adaptive strategy is executed. Thus, the application status is "WAIT" from 1.5 to 229 s, $t = 336$ and 351 s. The humidity gradually increases and stabilizes after 352.5 s. Humidity is above the requirement threshold at 610 s; hence, the application executes adaptive strategy #8 (i.e., turns off the humidifier) and #9 (i.e., opens the window). As a result of the adaptive strategies, humidity decreases and stabilizes from 628 to 744 s. Humidity decreases at 745 s; accordingly, the application extracts strategies (i.e., strategies #6 and #10) and executes them to satisfy the requirement. Under strategy #6, the humidifier is turned on to increase the indoor humidity. Under strategy #10, the window is closed to maintain indoor humidity. After adaptations, the humidity is stabilized again after 831 s.

Fig. 23 shows the results for Scenario #3. Under this scenario, adaptation is simultaneously required for both requirements, and there is a strategy for that. Initially, the light requirement should be increased, and the humidity requirement should decrease. The application selects strategy #11 for adaptation. Then, the humidifier is turned off, and the lamp power is increased. After the execution of strategy #11, the

TABLE IV
EXTRACTED STRATEGIES

Strategy	**Classification of strategies**	**Strategy score / Strategy**
Strategy #1	Optimal Solution	* 0.89 / Light(increase)
	Candidate Strategies	* 0.80 / Window(close)
Strategy #2	Optimal Solution	* 0.89 / Light(increase)
	Candidate Strategies	* 0.80 / Window(open)
Strategy #3	Optimal Solution	* 0.89 / Light(increase)
	Candidate Strategies	* No candidate Strategy
Strategy #4	Optimal Solution	* 0.89 / Light(decrease)
	Candidate Strategies	* 0.80 / Window(close)
Strategy #5	Optimal Solution	* 0.89 / Light(decrease)
	Candidate Strategies	* No candidate Strategy
Strategy #6	Optimal Solution	* 0.89 / Humidifier(on)
	Candidate Strategies	* 0.80 / Window(close)
Strategy #7	Optimal Solution	* 0.89 / Humidifier(on)
	Candidate Strategies	* No candidate Strategy
Strategy #8	Optimal Solution	* 0.89 / Humidifier(off)
	Candidate Strategies	* 0.80 / Window(open)
Strategy #9	Optimal Solution	* 0.80 / Window(open)
	Candidate Strategies	* No candidate Strategy
Strategy #10	Optimal Solution	* 0.80 / Window(close)
	Candidate Strategies	* No candidate Strategy
Strategy #11	Optimal Solution	* 0.80 / Humidifier(off) & Light(increase)
	Candidate Strategies	* 0.73 / Light(increase) & Window(open) * 0.73 / Humidifier(off) & Window(close)
Strategy #12	Optimal Solution	* 0.89 / Window(open)
	Candidate Strategies	* 0.73 / Light(decrease) & Window(open)
Strategy #13	Optimal Solution	* 0.73 / Light(increase) & Window(open)
	Candidate Strategies	* No candidate Strategy
WAIT	Unsatisfied requirement and wait for adaptation	

application extracts new strategies to satisfy the requirements, and strategy #2 is selected for adaptation. Under strategy #2, the lamp power is increased, and the window is opened. Opening the window increases the brightness and decreases humidity because the outdoor light is brighter than the indoor light, and the outdoor humidity is lower than the indoor humidity. In addition, strategy #2 increases the lamp power, thus increasing the indoor brightness. This shows that the extracted strategy can adapt to a specific situation. The light requirement was satisfied at 9 s, and the humidity requirement stabilized at 90 s. The illumination of the environment suddenly decreased from 188 to 134 lx at 169.5 s. The application extracts a strategy to adapt to environmental changes and increases the lamp power (i.e., strategy #1). In addition, the application executes strategy #1 at 310.5 and 324 s. The humidity decreases at 249 s; hence, the humidifier is turned on (i.e., strategy #6). The application closes the window at 256.5, 258.5, 274.5, and 295.5 s to maintain the indoor humidity level. The humidity requirement stabilizes after 297 s. This experiment shows that the application can adapt not only for a single requirement but also for multiple requirements.

Fig. 24 shows the results for Scenario #4. The light requirement is satisfied, but the humidity requirement is not satisfied at 0 s. The application extracts strategies and turns off the humidifier (i.e., strategy #8). The humidity requirement should still be decreased; thus, strategy #9 is extracted and executed (i.e., the window opens to decrease humidity). However,

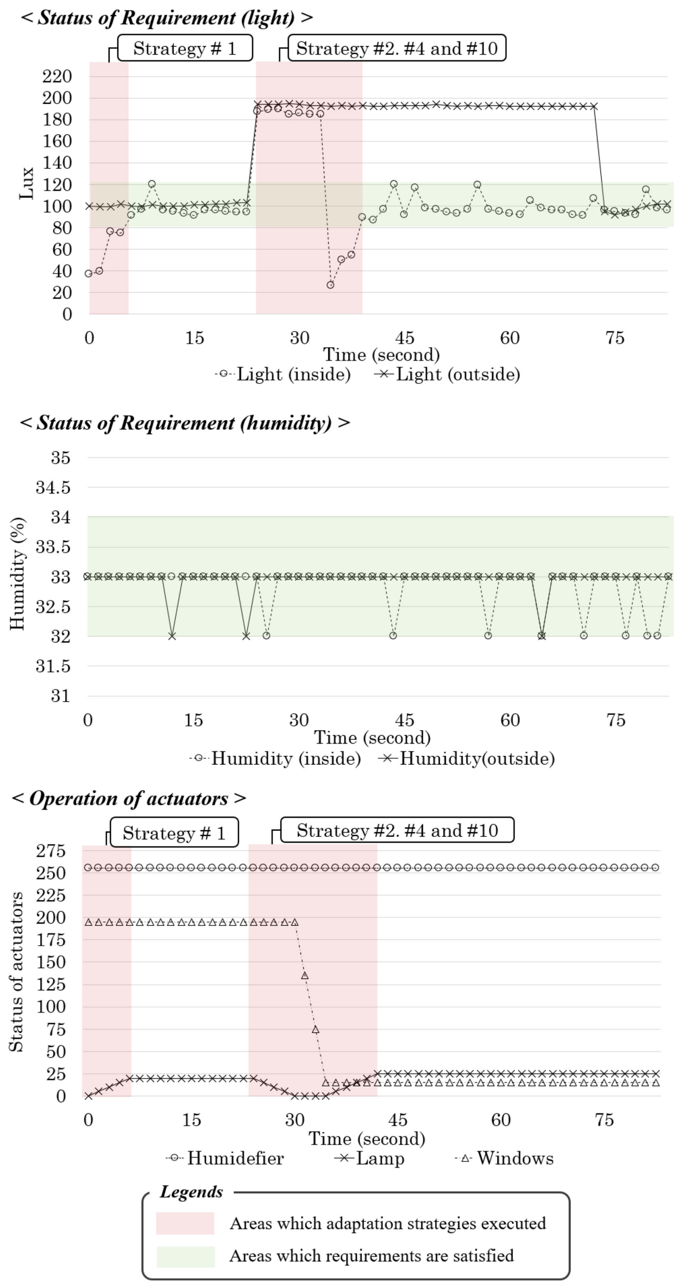


Fig. 21. Results of scenario #1.

strategy #9 causes environmental changes with respect to the light requirement because outdoor light affects the intensity of the indoor brightness. Hence, the application extracts a strategy for the light requirement, and strategy #5 is executed from 13.5 to 54 s and 99 s. The illumination of the environment suddenly decreases from 177 to 109 lx at 207 s. To adapt to the environmental change, the application extracts strategy #1, and the lamp power is increased from 207 to 250 s. Strategy #1 has a candidate strategy (i.e., the window is closed), but it affects humidity. Thus, the candidate strategy was not selected at that time. At 370.5 s, the light requirement should be increased; hence, the application executes strategy #3 (i.e., the lamp power is increased, and there is no candidate strategy). Humidity decreases at 306 s; hence, the humidifier is turned on (i.e., strategy #6). The application

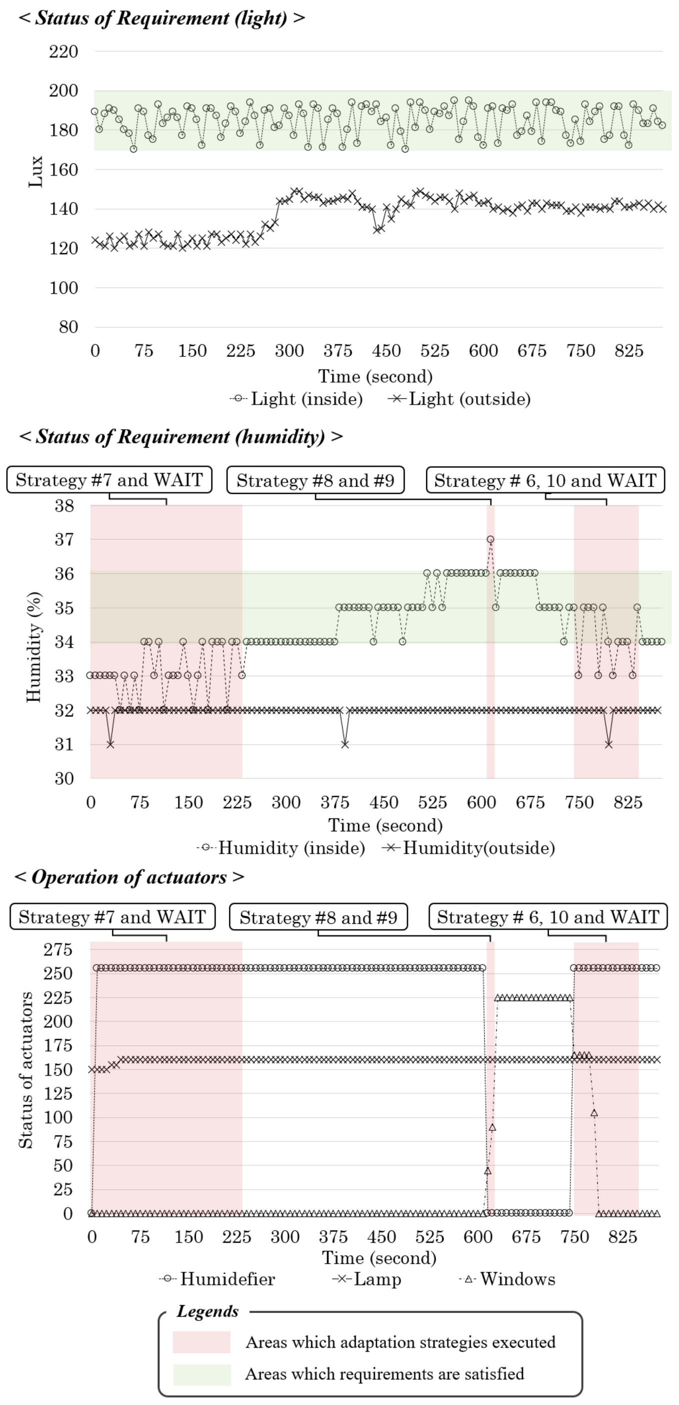


Fig. 22. Results of scenario #2.

closes the window at 313.5, 324, 328.5, 333, and 334.5 s to maintain indoor humidity levels. The humidity requirement stabilizes after 388.5 s. This experiment showed that an adaptive strategy could result in another adaptation. However, the experimental results demonstrate that the application can adapt to various environmental changes.

The experimental application shows an example of how the proposed framework worked. In addition, the results of the scenarios show that the proposed framework performs reasonably well with regard to adaptation to various environmental changes. Overall, the results of the scenarios demonstrate the

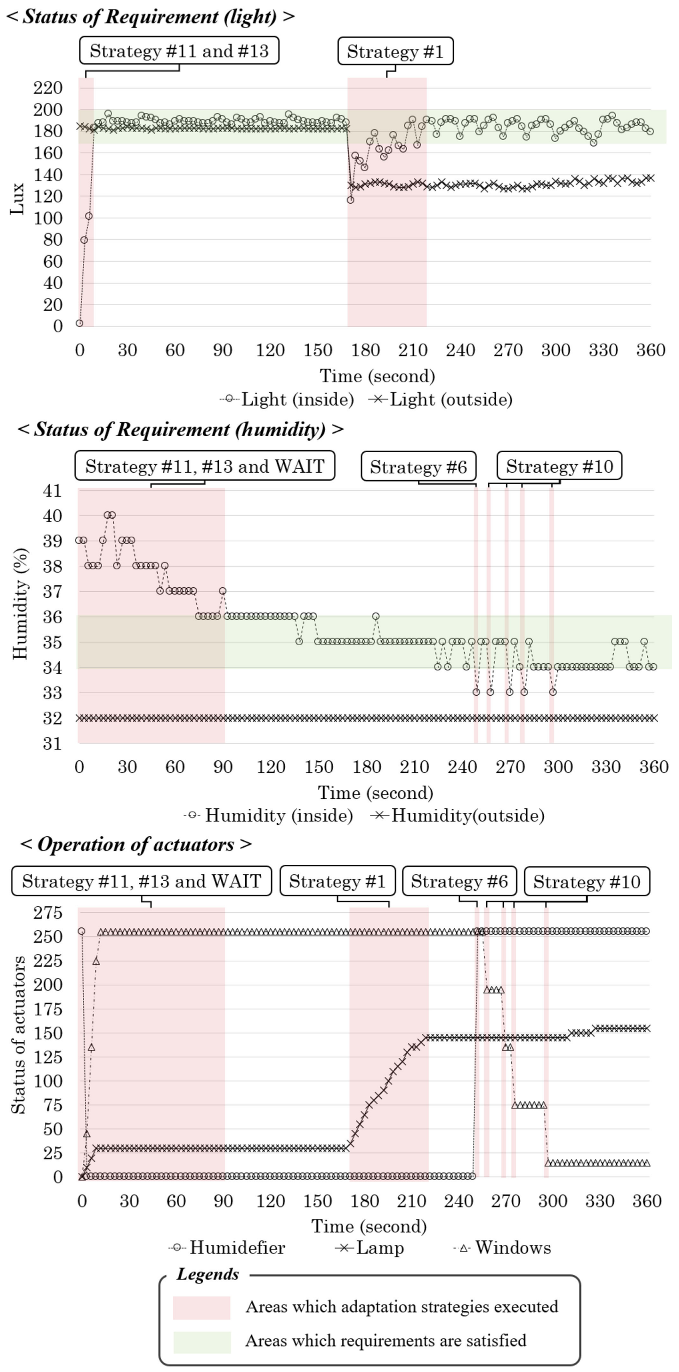


Fig. 23. Results of scenario #3.

effectiveness of the proposed self-adaptive framework for IoT at runtime.

VI. Discussion

In this study, we propose a self-adaptive software framework with a hierarchical finite-state machine for IoT systems. The proposed framework yielded excellent results. However, several problems must be solved before moving forward. In this section, we present topics on the limitations of the proposed framework and possible future work to overcome these limitations.

Predefined information is required to apply the proposed framework. For example, the framework can design a

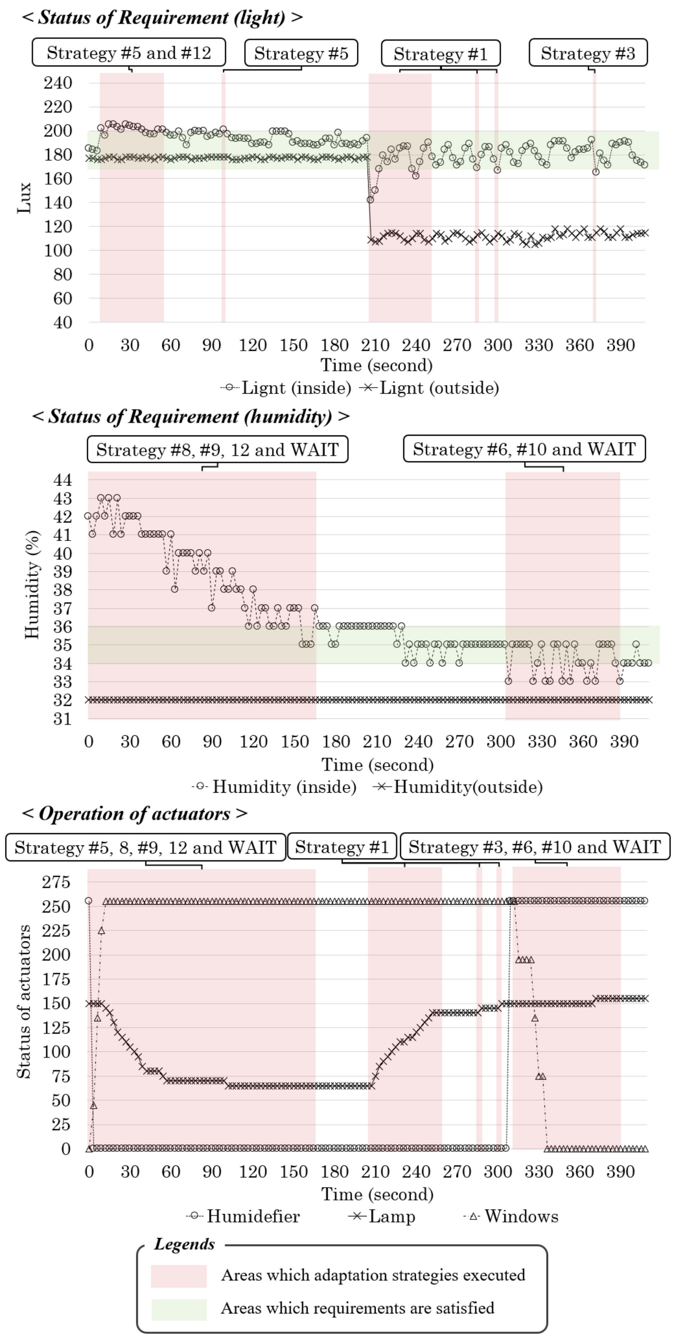


Fig. 24. Results of Scenario #4.

finite-state machine with information on IoT device capability, and the relationships among sensors, actuators, and requirements are defined. Therefore, various types of information must be defined and built-in IoT entities (e.g., sensors, actuators, users, and requirements). In addition, it may cause interoperability problems when applying the framework in the IoT system. However, several international organizations (e.g., ISO/IEC, IEEE Standards, ITU-T, oneM2M, IETF, and OCF) have developed IoT-related standards, and various topics related to interoperability exist (e.g., interworking, architecture, syntactic, semantic, and framework) [1]. Therefore, to overcome these limitations, international standards can be applied to the proposed framework to solve the interoperability and description of IoT entities.

It is the ultimate challenge that the self-adaptive system is minimizing human involvement [50], and this topic has been studied until recently [62]. The proposed framework also requires human involvement. In the proposed framework, the user checks the system status and provides additional information in the proposed framework. In addition, the proposed approach designs a finite-state machine (i.e., ML-FSM) with predefined information. However, predicting all interactions among various IoT entities is challenging, and the prediction may differ, as expected. To minimize human involvement and enhance system design, methods to extract meaningful information, such as user requirements and relationships among IoT devices, are required. Therefore, it is necessary to automatically generate the relationships among IoT entities. To overcome this limitation, we planned to apply a data-driven self-adaptive system with machine-learning-based approaches.

In this study, we focused on IoT environments in which IoT devices and requirements are closely related; thus, the operations of the actuators can affect sensed values and requirement satisfaction. Therefore, the framework was optimized for a specific IoT architecture style. The proposed framework is based on the master/slave architecture; thus, the master layer (i.e., host device) can recognize relationships among IoT devices and requirements and extract adaptation strategies using the relationship. In addition, a master/slave architecture is required for centralized management. However, there are various IoT environments, and these environments are considered to support self-adaptation. To overcome this limitation, the proposed approach can be extended with different IoT architecture styles [60] (e.g., centralized, information sharing, top-coordinated, regional planning, master/slave, and hierarchical), and this extension will be investigated in the future.

Moreover, a greenhouse application was introduced (see Section V) to show the working of the proposed framework. The purpose of the application was to enhance the understandability of the proposed framework (e.g., how modeling works, how the model is generated, and how the adaptation is performed). Therefore, the application was considered a simple scenario because a complex scenario could interrupt the understanding of the proposed framework. However, in Section IV, we performed an empirical evaluation, and the results showed that the proposed approach can perform stable verification within a reasonable time. In addition, the performance of the game-theory-based strategy extraction shows a reasonable amount of time with complex IoT environments in prior research [11]. Therefore, the proposed approach can be applied to complex IoT environments with complex relationships between devices and requirements.

Cache-based model checking is a verification method in the proposed framework, and this method provides reasonable verification performance at runtime. For runtime verification, the method mainly focused on the reachability of the finite-state machine because the IoT model (i.e., AML-FSM) was designed to verify the models with the reachability of specific states. However, the other model checkers (i.e., CadenceSMV, NuSMV, and nuXmv) are designed to consider not only reachability but also other specifications with temporal logics (i.e., CTL and LTL) that might affect the verification performance time. The temporal logic provides richer expressions of requirements; thus, the logic helps in the analysis of IoT systems from various perspectives. Additionally, a more detailed requirement description is helpful for rich expressions of the requirements using temporal logic. To solve the limitations related to the expressions of the IoT model and requirements, we have plans to improve the design of the finite-state machine by reflecting various IoT characteristics, and the cache-based model-checking method will be enhanced to a model checker with temporal logic.

VII. Conclusion

In this study, a self-adaptive software framework was proposed using a hierarchical finite-state machine. The proposed finite-state machine models are used to design and verify IoT systems for IoT environments in which devices and requirements are closely related. The proposed approach consisted of two layers: 1) slave and 2) master. Both layers were designed using a finite-state machine (i.e., ML-FSM, SN-FSM, and ACT-FSM). The slave layer comprised two types of IoT devices: 1) sensors and 2) actuators; thus, the slave layer was responsible for sensing environmental factors and operations. The master layer was responsible for designing the model using the collected slave devices and managing the designed IoT system at runtime. In particular, the master layer included the MAPE loop for runtime verification and adaptation. In the MAPE loop, the IoT system and environments were monitored, and the IoT system was verified using the designed modeling results (i.e., AML-FSM) and cache-based model-checking method. If the verification results indicated that there was an unsatisfied requirement, game-theory-based strategy extraction was performed, and the optimal solution was executed by operating the actuators.

Empirical experiments were performed, and the results highlighted the superiority of the proposed approach compared to other tools (NuSMV, Cadence SMV, nuXmv, and RINGA). In addition, the results showed that the proposed approach can be applied to a complex IoT environment with a complicated relationship between IoT devices and requirements. In addition, the IoT-based smart greenhouse application was implemented to demonstrate the efficacy of the framework in a physical environment. The application results demonstrate the effectiveness of the proposed framework. Additionally, a discussion is presented which includes the limitations and future work. In the future, the proposed framework will be enhanced to address the discussed limitations.

References

[1] E. Lee, Y.-D. Seo, S.-R. Oh, and Y.-G. Kim, "A survey on standards for interoperability and security in the Internet of Things," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 2, pp. 1020–1047, 2nd Quart., 2021.

[2] *Information Technology—Internet of Things (IoT)—Vocabulary*, ISO/IEC Standard 20924:2021, Dec. 2018.

[3] A. Burger, C. Cichiwskyj, S. Schmeißer, and G. Schiele, "The elastic Internet of Things—A platform for self-integrating and self-adaptive IoT-systems with support for embedded adaptive hardware," *Future Gener. Comput. Syst.*, vol. 113, pp. 607–619, Dec. 2020.

[4] A. Sacco, M. Flocco, F. Esposito, and G. Marchetto, "An architecture for adaptive task planning in support of IoT-based machine learning applications for disaster scenarios," *Comput. Commun.*, vol. 160, pp. 769–778, Jul. 2020.

[5] R. M. Andrade, B. R. A. Junior, P. A. M. Oliveira, M. E. Maia, W. Viana, and T. P. Nogueira, "Multifaceted infrastructure for self-adaptive IoT systems," *Inf. Softw. Technol.*, vol. 132, Jul. 2021, Art. no. 106505.

[6] F. Alkhabbas, I. Murturi, R. Spalazzese, P. Davidsson, and S. Dustdar, "A goal-driven approach for deploying self-adaptive IoT systems," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Salvador, Brazil, 2020, pp. 146–156.

[7] R. Nawaratne, D. Alahakoon, D. De Silva, P. Chhetri, and N. Chilamkurti, "Self-evolving intelligent algorithms for facilitating data interoperability in IoT environments," *Future Gener. Comput. Syst.*, vol. 86, pp. 421–432, Sep. 2018.

[8] M. A. Serhani, H. T. El-Kassabi, K. Shuaib, A. N. Navaz, B. Benatallah, and A. Beheshti, "Self-adapting cloud services orchestration for fulfilling intensive sensory data-driven IoT workflows," *Future Gener. Comput. Syst.*, vol. 108, pp. 583–597, Jul. 2020.

[9] 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. Symp. Appl. Comput.*, 2017, pp. 65–68.

[10] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D.-K. 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.

[11] 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.*, pp. 1–9, Jun. 2019, doi: 10.1007/s12652-019-01367-2.

[12] 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.

[13] R. Casado-Vara, P. Chamoso, F. De la Prieta, J. Prieto, and J. M. Corchado, "Non-linear adaptive closed-loop control system for improved efficiency in IoT-blockchain management," *Inf. Fusion*, vol. 49, pp. 227–239, Sep. 2019.

[14] S. Y. Shin, S. Nejati, M. Sabetzadeh, L. C. Briand, C. Arora, and F. Zimmer, "Dynamic adaptation of software-defined networks for IoT systems: A search-based approach," in *Proc. IEEE/ACM 15th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst.*, 2020, pp. 137–148.

[15] M. A. Scrugli, D. Loi, L. Raffo, and P. Meloni, "A runtime-adaptive cognitive IoT node for healthcare monitoring," in *Proc. 16th ACM Int. Conf. Comput. Front.*, 2019, pp. 350–357.

[16] J. Van Der Donckt, D. Weyns, F. Quin, J. Van Der Donckt, and S. Michiels, "Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals," in *Proc. IEEE/ACM 15th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst.*, 2020, pp. 20–30.

[17] J. Cámara, H. Muccini, and K. Vaidhyanathan, "Quantitative verification-aided machine learning: A tandem approach for architecting self-adaptive IoT systems," in *Proc. IEEE Int. Conf. Softw. Archit. (ICSA)*, Salvador, Brazil, 2020, pp. 11–22.

[18] H. Muccini and K. Vaidhyanathan, "Leveraging machine learning techniques for architecting self-adaptive IoT systems," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, Bologna, Italy, 2020, pp. 65–72.

[19] A. Pauna, I. Bica, F. Pop, and A. Castiglione, "On the rewards of self-adaptive IoT honeypots," *Ann. Telecommun.*, vol. 74, no. 7, pp. 501–515, 2019.

[20] H. Ouechtati, N. B. Azzouna, and L. B. Said, "Towards a self-adaptive access control middleware for the Internet of Things," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Chiang Mai, Thailand, 2018, pp. 545–550.

[21] H. Muccini, R. Spalazzese, M. T. Moghaddam, and M. Sharaf, "Self-adaptive IoT architectures: An emergency handling case study," in *Proc. 12th Eur. Conf. Softw. Archit. Companion*, 2018, pp. 1–6.

[22] Y.-D. Seo, Y.-G. Kim, E. Lee, K.-S. Seol, and D.-K. Baik, "Design of a smart greenhouse system based on MAPE-K and ISO/IEC-11179," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Las Vegas, NV, USA, 2018, pp. 1–2.

[23] H. Kim, E. Lee, and D.-K. Baik, "Self-adaptive software simulation: A lighting control system for multiple devices," in *Proc. Asian Simul. Conf.*, 2017, pp. 380–391.

[24] E. Lee, Y.-D. Seo, and Y.-G. Kim, "A cache-based model abstraction and runtime verification for the Internet-of-Things applications," *IEEE Internet Things J.*, vol. 7, no. 9, pp. 8886–8901, Sep. 2020.

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

[26] "NUSMV: A New Symbolic Model Checker." [Online]. Available: http://nusmv.fbk.eu/ (accessed Aug. 31, 2020).

[27] A. Cimatti *et al.*, "NUSMV 2: An OpenSource tool for symbolic model checking," in *Proc. Int. Conf. Comput.-Aided Verif.*, 2002, pp. 359–364.

[28] "The NUXMV Model Checker." [Online]. Available: https://nuxmv.fbk.eu/ (accessed Aug. 31, 2020).

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

[30] E. M. Clarke Jr., O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*. Cambridge, MA, USA: MIT Press, 2018.

[31] 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.

[32] 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.

[33] B. Schmerl *et al.*, "Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations," in *Proc. Symp. Bootcamp Sci. Security*, 2014, pp. 1–12.

[34] 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.

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

[36] 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.

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

[38] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Heidelberg, Germany: Springer, 2011, pp. 1–30.

[39] 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.

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

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

[42] 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.

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

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

[45] 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.

[46] 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.

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

[48] G. Su, T. Chen, Y. Feng, and D. S. Rosenblum, "ProEva: Runtime proactive performance evaluation based on continuous-time Markov chains," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, Buenos Aires, Argentina, 2017, pp. 484–495.

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

[50] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.

[51] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[52] J. Ariza, K. Garcés, N. Cardozo, J. P. R. Sánchez, and F. J. Vargas, "IoT architecture for adaptation to transient devices," *J. Parallel Distrib. Comput.*, vol. 148, pp. 14–30, Feb. 2021.

[53] G. Schiele, A. Burger, and C. Cichiwskyj, "The elastic node: An experimentation platform for hardware accelerator research in the Internet of Things," in *Proc. IEEE Int. Conf. Auton. Comput. (ICAC)*, Umea, Sweden, 2019, pp. 84–94.

[54] D. Alahakoon, S. K. Halgamuge, and B. Srinivasan, "Dynamic self-organizing maps with controlled growth for knowledge discovery," *IEEE Trans. Neural Netw.*, vol. 11, no. 3, pp. 601–614, May 2000.

[55] D. De Silva and D. Alahakoon, "Incremental knowledge acquisition and self learning from text," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Barcelona, Spain, 2010, pp. 1–8.

[56] M. U. Iftikhar, G. S. Ramachandran, P. Bollansée, D. Weyns, and D. Hughes, "DeltaIoT: A self-adaptive Internet of Things exemplar," in *Proc. IEEE/ACM 12th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, Buenos Aires, Argentina, 2017, pp. 76–82.

[57] F. M. Barreto, P. A. D. S. Duarte, M. E. F. Maia, R. M. D. C. Andrade, and W. Viana, "CoAP-CTX: A context-aware CoAP extension for smart objects discovery in Internet of Things," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1. Turin, Italy, 2017, pp. 575–584.

[58] M. E. F. Maia, A. Fonteles, B. Neto, R. Gadelha, W. Viana, and R. M. C. Andrade, "LOCCAM—Loosely coupled context acquisition middleware," in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, 2013, pp. 534–541.

[59] B. R. A. Junior, R. M. C. Andrade, M. E. F. Maia, and T. P. Nogueira, "Succeed: Support mechanism for creating and executing workflows for decoupled SAS in IoT," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2. Tokyo, Japan, 2018, pp. 738–743.

[60] H. Muccini and M. T. Moghaddam, "IoT architectural styles," in *Proc. Eur. Conf. Softw. Archit.*, 2018, pp. 68–85.

[61] "Arduino—Home." [Online]. Available: https://www.arduino.cc/ (accessed Aug. 12, 2021).

[62] Y. Elkhatib and A. Elhabbash, "If a system is learning to self-adapt, who's teaching?" in *Proc. Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, Madrid, Spain, 2021, pp. 256–257.

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 was a Postdoctoral Researcher with the Department of Computer and Information Security, Sejong University, Seoul. He is currently an Assistant Professor with the School of Computer Science, Chungbuk National University, Cheongju, South Korea. 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, Republic of 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 was an Assistant Professor with the Department of Data Science. He is currently an Assistant Professor with the Department of Computer Engineering, Inha University, Incheon, South Korea. 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 (minor 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, and Convergence Engineering for Intelligent Drone, Sejong University, Seoul. He has published over 130 research papers in the field of computer science and information security. As a Korean ISO/IEC JTC 1 Member, he is contributing in developing data exchange standards. His current research interests include big data security, network security, home network, security risk analysis, and security engineering.