Information and Software Technology 93 (2018) 200–222

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

RINGA: Design and verification of finite state machine for self-adaptive software at runtime

Euijong Lee[a,*], Young-Gab Kim[b,**], Young-Duk Seo[a,*], Kwangsoo Seol[a,*], Doo-Kwon Baik[a,*]

[a] Department of Computer Science and Engineering, Korea University, Seoul, Republic of Korea
[b] Department of Computer and Information Security, Sejong University, Seoul, Republic of Korea

A R T I C L E I N F O

Keywords:
Self-adaptive software
Model checking
Finite state machine
Runtime

A B S T R A C T

Context: In recent years, software environments such as the cloud and Internet of Things (IoT) have become increasingly sophisticated, and as a result, development of adaptable software has become very important. Self-adaptive software is appropriate for today's needs because it changes its behavior or structure in response to a changing environment at runtime. To adapt to changing environments, runtime verification is an important requirement, and research that integrates traditional verification with self-adaptive software is in high demand.
Objective: Model checking is an effective static verification method for software, but existing problems at runtime remain unresolved. In this paper, we propose a self-adaptive software framework that applies model checking to software to enable verification at runtime.
Method: The proposed framework consists of two parts: the design of self-adaptive software using a finite state machine and the adaptation of the software during runtime. For the first part, we propose two finite state machines for self-adaptive software called the self-adaptive finite state machine (SA-FSM) and abstracted finite state machine (A-FSM). For the runtime verification part, a self-adaptation process based on a MAPE (monitoring, analyzing, planning, and executing) loop is implemented.
Results: We performed an empirical evaluation with several model-checking tools (i.e., NuSMV and CadenceSMV), and the results show that the proposed method is more efficient at runtime. We also investigated a simple example application in six scenarios related to the IoT environment. We implemented Android and Arduino applications, and the results show the practical usability of the proposed self-adaptive framework at runtime.
Conclusions: We proposed a framework for integrating model checking with a self-adaptive software lifecycle. The results of our experiments showed that the proposed framework can achieve verify self-adaptation software at runtime.

1. Introduction

Nowadays, various software platforms (e.g., smartphone operating systems, cloud environments, Arduino, Raspberry Pi, and web-based applications) are available. As a result, various software applications depending on platforms such as the cloud and Internet of Things (IoT)[1] have become widespread. Furthermore, rapid advancements in mobile and IoT devices have led to a demand in software systems that can operate in various environments. Hence, self-adaptive software is software that changes its behavior or structure in a changing environment at runtime [1]. Self-adaptive software satisfies the current need for software that can operate in various environments. Verification is one of the most important tasks for self-adaptive software, and it needs to be performed at runtime [1]. To viably support runtime verification, the integration of traditional verification with self-adaptive software is preferable [2,3]. Model checking is an effective static verification method for software and is governed by the state-based model [4]. Despite excellent verification performance, model checking incurs a problem at runtime: state explosion [5]. Therefore, model checking needs to be integrated in the self-adaptation lifecycle during runtime verification.

There are several studies [6–13] on state machines and model checking in self-adaptive software. On one hand, some studies [6–8] apply state machines and model checking to self-adaptive software

* Corresponding author at: Department of Computer Science and Engineering, Korea University, Seoul, Republic of Korea.
** Corresponding author at: Department of Computer and Information Security, Sejong University, Seoul, Republic of Korea.
E-mail addresses: kongjjagae@korea.ac.kr (E. Lee), alwaysgabi@sejong.ac.kr (Y.-G. Kim), seoyoungd@korea.ac.kr (Y.-D. Seo), seolks@korea.ac.kr (K. Seol), baikdk@korea.ac.kr (D.-K. Baik).
[1] Full definition of acronyms is described in Annex.

http://dx.doi.org/10.1016/j.infsof.2017.09.008
Received 13 September 2016; Received in revised form 8 September 2017; Accepted 16 September 2017
Available online 20 September 2017
0950-5849/ © 2017 Elsevier B.V. All rights reserved.

design and evaluation. Such methods are useful during design-time verification and post-processing for evaluating self-adaptive software, but they have limitations when applied at runtime for verification. On the other hand, some studies [10–12] apply probabilistic model checking to verify self-adaptive software at runtime and use an interactive state machine (ISM) to verify self-adaptive software that suffers from uncertainty. An ISM is a state machine that can interactively update its model and requirements in response to environment changes. We assume that the runtime environment of self-adaptive software is not predictable, and therefore, an ISM is more suitable for verifying self-adaptive software at runtime. However, previous studies on ISMs have only been optimized to solve uncertainty problems. Therefore, in this study, we propose a finite state model to design and verify self-adaptive software.

In this paper, to resolve the above design and verification issues, we propose two types of finite state machines for the design and verification of self-adaptive software. Furthermore, we propose a self-adaptive framework called RINGA (Runtime verIfication with fiNite state machine desiGn for self-Adaptation software) using the two types of finite state machines. RINGA consists of two parts: the design-time part is responsible for the design of finite and abstract-state machines for performance at runtime, and the runtime part consists of a MAPE (Monitoring-Analyzing-Planning-Executing) loop that analyzes the environment with the abstract state machine extracted from the design-time part. We performed an empirical evaluation using symbolic model checking tools, and our results show that the proposed method can be used to perform verification at runtime. Furthermore, we implemented an Android and Arduino application with an IoT-based example application in six scenarios to measure the adaptability of the proposed framework.

The remainder of the paper is organized as follows. Section 2 provides background on self-adaptive software and work related to runtime model checking for self-adaptive software. Section 3 introduces the proposed framework. Section 4 presents the empirical evaluation. Section 5 presents the results of experiments with a simple IoT-based example application. Section 6 discusses the limitations of the proposed approach and extensions that could overcome these limitations. Section 7 provides the concluding remarks and discusses future work.

2. Related work

In this section, we introduce various self-adaptive frameworks and platforms, as well as previous studies on the verification requirements in self-adaptive software research. We summarize previous research and compare it with RINGA in Table 1. Details regarding the various studies are described in subsections on the frameworks, platforms, and verification of self-adaptive software.

2.1. *Self-adaptive software frameworks*

As mentioned earlier, to adapt to changing environments at runtime, self-adaptive software dynamically changes its behavior or structure [1]. Therefore, self-adaptive software detects the state of an environment and changes its behavior or structure if possible when its aim has been violated [1]. That is, self-adaptive software monitors its environment and analyzes the situation and environment to adapt to any changes. To accomplish this, the MAPE loop mechanism was proposed [1,14–16] and implemented in several self-adaptive software and autonomic computers. A MAPE loop consists of four parts:

- Monitoring: responsible for collecting and correlating data from the environment and internal software changes.
- Analyzing: responsible for analyzing the symptoms related to situation changes using the monitored data.
- Planning: determines the adaptive strategies, i.e., is responsible for determining what is to be changed and how.
- Executing: responsible for activating the adaptive strategies.

Table 1
Comparison of previous research and RINGA.

Work by	Goal	Lifecycle	Model for system design	Technique for verification	Experimental domain
Tesei et al. [6]	Self-adaptive system design and verification of constraint violations	N/A	State machine	Model checking	Ecology
Abeywickrama and Zambonelli [7]	Self-adaptive software design using a goal-based model	N/A	SOTA (a goal-based model)	Model checking	e-mobility
Johnson et al. [9]	Reverification of component-based software system	N/A	Probabilistic model	Model checking	Cloud services
Filieri et al. [10–12]	Statically generated verification conditions at runtime	Control loop	Discrete-time Markov chain	Model checking	Server–client web and lower wireless buses
Yang et al. [13]	Verification of self-adaptive software involving uncertainty in environmental interactions	Reaction loop	ISM	Model checking	Robot cars
Tallabaci and Souza [17]	Adaptive system design using a goal-based model	MAPE loop	Goal-based model	Checking fulfillment of goal-based model	Automated teller machines
Barna et al. [18]	Self-adaptive cloud based exemplar	MAPE loop	N/A	N/A	Cloud environments
Garlan et al. [19]	Support reusable infrastructure	Control loop	Architecture	Software architecture and reusable infrastructure	Web-based client–server systems
Knauss et al. [21]	Adaptation of contextual requirements	MAPE loop	Contextual requirement	Machine learning, data mining	Activity-scheduling systems
Wuttke et al. [23]	Self-adaptive traffic routing based exemplar	N/A	N/A	N/A	Traffic routing
Weyns and Calinescu [24]	Self-adaptive service-based system exemplar	MAPE loop	N/A	N/A	Tele-assistance systems
RINGA	**Self-adaptive software design and verification at runtime**	**MAPE loop**	**SA-FSM, A-FSM**	**Model checking**	**Light control applications**

As described above, the MAPE loop contains the general process of self-adaptive software and autonomic computing. Therefore, several self-adaptive software frameworks have been constructed using this loop [1,7,9–12,17–21].

To construct self-adaptive software, various self-adaptive software frameworks have been developed recently, and these frameworks each have their own aspects and distinct characteristics [1,7,9–12,17–21]. For example, Rainbow [19] supports self-adaptation using reusable infrastructure and software architecture. This framework monitors the properties of a running system and evaluates architecture models to find constraint violations. This framework uses external adaptation mechanisms to adapt to multiple system concerns. Similarly, INVEST (Incremental VErification STrategy) [9] uses components to describe software, and this framework adapts by adding, removing, or modifying the components. This framework verifies its status using incremental verification by component model checking; it also guarantees the verification using probabilistic results. Knauss et al. [21] proposed a self-adaptive framework to cope with uncertainty in contextual requirements at runtime called ACon. ACon is designed using the MAPE loop and updates contextual requirements using data mining and machine learning (i.e., rule-based classifiers) at runtime.

In addition, there are goal model-based frameworks [7,17]. Tallabaci and Silva Souza [17] designed an adaptive system from the perspective of requirements engineering and feedback loop called Zanshin. This framework is designed based on requirements using a goal model [22] and verifies software by checking the fulfillment of the goal model. If a sub-goal is violated, Zanshin executes the related adaptive strategy. Furthermore, State of the Affairs (SOTA) [7] is a goal-oriented modeling framework that designs self-adaptive software using a goal model, and then translates the goal model into a state-based model. The translated state-based model is used to verify software.

There are other state-based frameworks similar to SOTA [6,12,13]. Tesei et al. [6] proposed a multi-level finite state machine that is classified into two types of adaptations: structural (i.e., related to architectural reconfiguration) and behavioral (i.e., related to functional changes). Structural adaptation is an upper-level adaptation, where the lower level (i.e., the behavioral level) is modeled as a state machine and the upper level is modeled as a second-order state machine. Using such multiple levels, this framework's upper model verifies constraint violations at lower models. Yang et al. [13] proposed a state machine model for verifying self-adaptive applications that involve uncertainty in environmental interactions. The researchers called the proposed state model the ISM, and it is based on a probabilistic state machine. This framework conducts verification using counter-examples prioritized according to ISM probabilities. Filieri et al. [12] proposed a discrete-time Markov chain model that has a checking-based framework for self-adaptive software. This framework provides static verification conditions for evaluating environments at runtime. In addition, it provides sensitivity analysis by reasoning about the changes of effectors and extracts adaptation strategies.

In this paper, we follow the MAPE loop at runtime, similar to previous research. MAPE applies model-checking techniques to verify self-adaptive software. There are various methods for designing self-adaptive software, as described in this section. We assume that finite state machines are useful for designing self-adaptive software and for verification at runtime using model-checking techniques [4]. Therefore, the proposed framework builds two types of finite state machines: one is responsible for designing the software and the other is responsible for performing verification at runtime. The proposed framework is described in detail in Sections 3.1 and 3.2.

2.2. *Self-adaptive software platforms*

There are various platforms for performing or simulating self-adaptive software. Web-based platforms are widely used in self-adaptive studies [9–12,18,19], and they are classified into two types: server-client-based [10–12,19] and cloud service-based [9, 18]. In these studies, the platforms are implemented and simulated. To prove adaptability with web-based platforms, the proposed method is compared with a non-adaptive implementation [9] and other non-adaptive methods [10–12] or case studies [18,19]. Some studies are related to solving car traffic [23] and parking [7] problems. Car traffic and parking platforms cannot be easily implemented in the real world, and thus simulations are performed to prove adaptability.

Some studies have their own platforms, such as eco-systems [6], automated teller machines [17], robot cars [13], service-based systems [24], and activity scheduling systems for the sport of rowing in unpredictable environments (i.e., ToTEM) [21]. In such cases, studies focus on simulating software in the real world [13,21], simulating software based on case studies [17], or verifying compatibility using scenarios [6,24] to show adaptability. Overall, previous studies have used independent platforms and proved their adaptability through reasonable experiments. One of the problems of self-adaptive fields is that there are no prototypical applications that researchers can use to evaluate or compare new methods [23,24]. To solve this problem, studies have provided common scenarios, simulation tools, and prototypes [18,23,24]. However, such studies are rarely employed because there are various aspects to self-adaptive software, and these aspects depend on the platform.

In this paper, we aim to support runtime verification for finite state machines while considering devices with low computing power. To execute the proposed method in the real world, we devised a simple IoT-based platform (Section 5). This simple platform demonstrates the adaptability of the proposed framework in six scenarios (Section 5.3).

2.3. *Verification of self-adaptive software*

Runtime verification is an important requirement of self-adaptive software [1–3]. Previous studies on roadmaps [2] suggest that traditional verification and validation techniques need to be integrated into the self-adaptation software lifecycle. Model checking is a verification and validation technique that can be adapted to the self-adaptive software lifecycle. Furthermore, model checking enables runtime verification because this method can support verification and investigative mechanisms [1,2]. However, model checking has limitations when used for runtime verification because of its inherent problems, such as state explosion [1,2,4,5]. Therefore, model checking has been modified for the self-adaptive software lifecycle. Studies have applied state machines or model checking to self-adaptive software with different aspects and uses [6–13].

Moreover, there are studies that apply model checking to self-adaptive software during the design phase for evaluating the software at design time [6–8]. Tesei et al. [6] proposed a multiple-level state machine for self-adaptive software called the S[B]-system that is verified by model checking to confirm the designed self-adaptive software. Abeywickrama and Zambonelli [7]. developed a state machine as a goal-oriented model and used this state model (i.e., SOTA) to analyze and design self-adaptive software. This research verified the SOTA model for self-adaptive software analysis during the early stages of the self-adaptive software lifecycle. Cámara and De Lemos [8] applied probabilistic model checking to evaluate self-adaptive systems. This research collects data and creates a probabilistic model using the data. Using the results of probabilistic model checking, they provide levels of confidence with regard to service delivery. These studies apply model checking and state machine to self-adaptive software design and evaluation. However, the former are optimized during pre- and post-processes, and thus not suitable for runtime verification.

In addition, some studies have applied state machines and model checking at runtime for self-adaptive software. Yang et al. [13] proposed a state machine model to verify self-adaptive software that suffers from uncertainties in the environment called ISM. This model is verified at runtime, and it generates counter examples that are prioritized according to their probabilities. Johnson et al. [9] used probabilistic automata to verify self-adaptive software called INVEST. INVEST verifies the designed probabilistic automata to reverify the probabilistic safety properties of cloud-based software. Some studies [10–12] have proposed a framework that uses probabilistic model checking to verify self-adaptive software at runtime. The probabilistic model is precomputed and translated as a function of the expression used to verify software at runtime. These studies apply probabilistic state machines [9–12] or condition-based models [13]. We assume that the probabilistic approach has inherent problems because an environment's conditions are not predictable by self-adaptive software at runtime. Therefore, we propose modeling environment conditions based on finite state machines for the design and verification of self-adaptive software (Section 3.2.1). As mentioned earlier in this section, model checking has inherent problems [1,2,4], and thus we refer to previous studies that use precomputed state machines [10–12]. The proposed framework extracts a runtime model (i.e., A-FSM) from a designed software model (i.e., SA-FSM) for runtime verification (Section 3.2.2). The runtime model is verified with a MAPE loop at runtime. The details of the proposed framework are provided in Section 3.

3. RINGA: self-adaptive software framework with runtime verification

We propose a self-adaptive software framework called RINGA for performing runtime verification using the finite state machine-based model checking. Section 3.1 presents an overview of RINGA, which consists of preprocessing with a finite state machine and a MAPE loop. Section 3.2 explains how to design a finite state machine for the proposed framework. Details on runtime verification with a MAPE loop are provided in Section 3.3.

3.1. Overview

The MAPE loop is an adaptation process mechanism for self-adaptive software [1,14–16]. As mentioned in Section 2.1, this closed-loop mechanism consists of four processes: monitoring, analyzing, planning, and executing. In this paper, we propose a self-adaptive software framework based on the MAPE loop. Fig. 1 shows an overview of the proposed framework, which consists of two parts: design-time and runtime.

The design-time part is responsible for designing a finite state machine model, extracting data to be monitored during the runtime phase, and abstracting the processes of the designed model. We propose rules for the finite state machine for self-adaptive software. The finite state machine has four state types: normal, satisfied, dissatisfied, and adaptive and two transition types: normal and adaptive. We call this finite state machine a SA-FSM. After designing the SA-FSM, it is evaluated to check that the model satisfies requirements and follows SA-FSM rules. This correctness evaluation process is important because if the designed SA-FSM is incorrect, it can incorrectly execute the adaptive strategy or not adapt to an unexpected event. If the designed SA-FSM is incorrect, the design of the finite state machine model is repeated. After the evaluation process, monitoring data and triggers are extracted. Monitoring data are related to the environment or software changes during software implementation, and triggers are factors for changing the software. After extracting the monitoring data and triggers, the designed SA-FSM is abstracted for runtime verification as an A-FSM. Model checking is a reliable method for verifying software, but it requires a significant amount of computing resources. Therefore, we present an abstraction algorithm for abstracting SA-FSM in the definition of A-FSM by applying state elimination algorithms [25]. The abstraction algorithm is executed once if there is no change in the SA-FSM; this prevents any impact at runtime. Consequently, after the design process is complete, the proposed framework produces an A-FSM based on the predesigned SA-FSM. The A-FSM is transferred to the runtime part and used for analysis in runtime verification. Furthermore, the A-FSM is used to evaluate the condition of a self-adaptive software in each cycle of MAPE loop at runtime. Details of the design-time process are further described in Section 3.2.

The runtime part consists of the MAPE loop mechanism, as depicted in Fig. 1. During the monitoring process, the values of the monitored data and trigger statuses are checked. Next, the analysis process calculates the equations that are described as transitions of the A-FSM. This process only calculates the equations, rather than model checking. Thus, it can be performed more rapidly than previous model-checking methods at runtime (Section 4.2). The A-FSM determines the probability that the software can adapt to the changing environment. Subsequently, the planning process selects a predefined adaptive strategy accordingly. In the planning process, an adaptive strategy is determined for each result that requires adaptation. The execution process implements the adaptive strategy selected in the planning process. The adaptive strategies activate triggers that can also adapt to changes in the environment. These trigger data are defined at design-time. The executing process modifies the trigger values to adapt to a changing environment. After execution, the monitoring process is executed again and the MAPE loop iterates. Furthermore, if the designed A-FSM can no longer adapt to the environmental changes and if there is time to design a new SA-FSM, the runtime part can request a model update from the design part. When a model update is requested, the predefined SA-FSM is redesigned to adapt to the environment changes to which the previous A-FSM failed to adapt. After the SA-FSM is redesigned, the evaluation, extraction, and abstraction processes are executed once, and the resultant A-FSM is transferred to the runtime part. Note that the redesigning of SA-FSM is only possible for systems where the time required to produce a new SA-FSM is acceptable.

In this study, we assume that the software should be implemented with reference to the designed model (SA-FSM and A-FSM). This means that each state is implemented in the software as a module (e.g., as an individual function). More specifically, each state needs to be implemented individually so that there is no interference between states. The proposed approach requires that each state be activated independently. In addition, transitions involve values to be monitored at runtime and triggers to activate adaptive strategies. Therefore, if the software and environment are changed, then the related transitions values in the SA-FSM change. In addition, if transitions related to the trigger value are changed in the SA-FSM, the triggers are activated in the software. In other words, the SA-FSM and software influence each other at runtime. To summarize, the proposed MAPE loop attempts to determine the situation so that it may adapt using simplified finite state model (i.e., A-FSM), which is extracted from the finite state machine generated for the self-adaptive software (i.e., SA-FSM). In the loop, if the A-FSM results show that there is a need to adapt to a changing environment, the software changes the triggers that cause it to adapt to the environment.

3.2. RINGA design-time process

In this section, we describe the design-time part of the proposed framework. Section 3.2.1 explains the rules for designing the SA-FSM. Section 3.2.2 explains how to abstract the SA-FSM to derive the A-FSM using state elimination rules.

3.2.1. SA-FSM design

We describe self-adaptive software as a finite state machine. The reason for this is to facilitate the reconstruction and verification when adaptation is required. As mentioned previously, we assume that self-adaptive software consists of modules responsible for performing a single operation (e.g., searching for connectable devices or connecting devices using Bluetooth networks). Therefore, a state can be described as a software module. We also assume that a transition operation is by

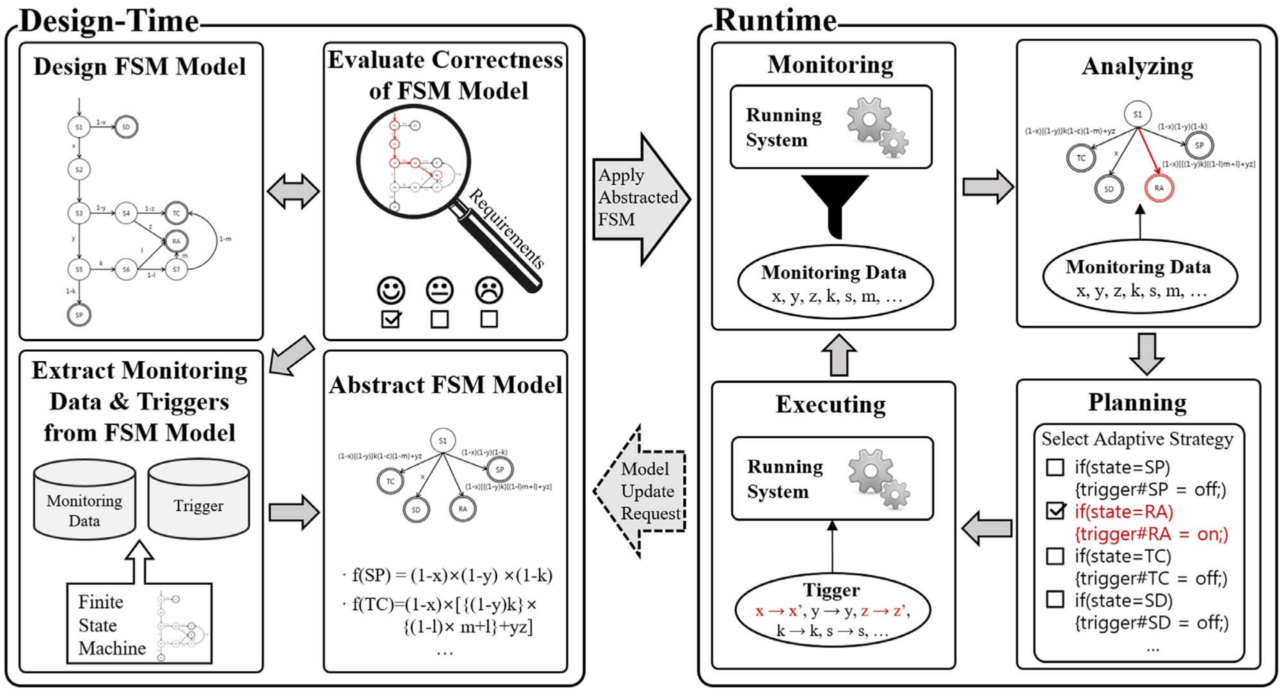

Design-Time
Design FSM Model
Evaluate Correctness of FSM Model
Requirements
Extract Monitoring Data & Triggers from FSM Model
Monitoring Data
Trigger
Finite State Machine
Abstract FSM Model
· f(SP) = (1-x)×(1-y) ×(1-k)
· f(TC)=(1-x)×[{(1-y)k}× {(1-l)× m+l}+yz]
…
Apply Abstracted FSM
Model Update Request
Runtime
Monitoring
Running System
Monitoring Data x, y, z, k, s, m, …
Analyzing
Monitoring Data x, y, z, k, s, m, …
Planning
Select Adaptive Strategy
if(state=SP) {trigger#SP = off;)
if(state=RA) {trigger#RA = on;)
if(state=TC) {trigger#TC = off;)
if(state=SD) {trigger#SD = off;)
…
Executing
Running System
Tigger
x → x', y → y, z → z', k → k, s → s, …

Fig. 1. Overview of RINGA.

monitored data or operation conditions. In summary, self-adaptive software operates according to the status of the related finite state machine. Self-adaptive software changes its behavior when it needs to adapt. Moreover, if a self-adaptive software program is described by a finite state machine, and each module matches with the corresponding states, reconstruction of the self-adaptive software is then realized by simply changing the transitions of the finite state machine. Furthermore, it is possible to add or delete a software module simply by modifying a state of the finite state machine if the module is matched with the state. In this case, we can reconstruct self-adaptive software by adding or deleting transitions and modules. In addition, by describing software as a finite state machine, we can apply various model-checking techniques to verify the self-adaptive software; this becomes even more significant because demand for practical verification at runtime has increased with regard to self-adaptive software. In this paper, to describe self-adaptive software, we propose the finite state machine called SA-FSM is a tuple (S, →, S_0, AP, L) where,

- S is a set of states,
- S consists of four subsets: $\{S_{normal}, S_{sat}, S_{dis}, S_{adaptive}\} \subseteq S$,
- $\rightarrow \subseteq S \times S$ is the transition relationship, and it is classified into two types: $\rightarrow_{normal}$ and $\rightarrow_{adaptive}$
- $\rightarrow_{adaptive} \subseteq \{S_{dis} \times S_{adaptive}\}$,
- $S_0 \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- L: $S \rightarrow 2^{AP}$ is a labeling function (2^{AP} denotes the power set of AP).

As represented by the tuple definition, SA-FSM consists of four sets of states and two transition types. The state sets are as follows:

- Normal state (S_{normal}): a set of states that do not impact software adaptation.
- Satisfied state (S_{sat}): the set of end states for which the software requirements are satisfied.
- Dissatisfied state (S_{dis}): the set of end states for which the software requirements are not satisfied. This type of state needs to be adapted for, and thus, after this state is entered, an adaptation trigger is required.
- Adaptive state ($S_{adaptive}$): set of states for which the adaptive activity is performed. If the software reaches this state, the related adaptive strategy is triggered.

The transition types are as follows:

- Normal transition ($\rightarrow_{normal}$): a normal transition that does not impact software adaptation. Some normal transitions need to be monitored at runtime.
- Adaptive transition ($\rightarrow_{adaptive}$): an adaptation trigger. When the software reaches a dissatisfied state, it checks the operating conditions of the related adaptive transition. If the corresponding condition is available, the related adaptive strategy is triggered.

For the overall process of the proposed framework, the self-adaptive software is designed by the SA-FSM during the first process of the design-time part (Section 3.1). A simplified model, A-FSM, is abstracted from SA-FSM. This simplified model is used to evaluate software condition in each MAPE loop. In other words, the design of SA-FSM is used at each MAPE loop (i.e., an A-FSM involves the design of an SA-FSM). Therefore, if the software reaches S_{sat}, a MAPE loop cycle is terminated, satisfying the software requirements. In addition, if the software reaches S_{dis}, a MAPE loop cycle is terminated, dissatisfying the software requirements; the software needs to be adapted accordingly. In this manner, the design of SA-FSM and A-FSM is used to validate the self-adaptive software in the MAPE loop at runtime. In Section 5.2.1, we describe an example that uses SA-FSM with a self-adaptive light control. In the next section, we introduce how to abstract a designed SA-FSM in the form of an A-FSM for verification at runtime.

3.2.2. A-FSM for runtime verification

After designing the SA-FSM, it is abstracted in the form of a simplified finite state machine (i.e., A-FSM) for verification at runtime. In this section, we explain the reason for changing an SA-FSM into an A-FSM, and how to perform this abstraction. As mentioned earlier, model checking is a powerful tool for verifying software. Furthermore, model checking can detect a violation of requirements and provide the cause. However, model checking has limitations such as state explosion when

```
1  // Input: SA-FSM
2  // Output: A-FSM
3
4  SA_FSMtoA_FSM(SA-FSM){
5     A-FSM = null;
6     while(SA-FSM.hasSsat){
7        tree = extractPath(SA-FSM.satState);
8        absTran = extractAbstractTran(tree);
9        A-FSM.addSsat(SA-FSM.satState, absTran);
10    }
11    while(SA-FSM.hasSdis){
12       tree = extractPath(SA-FSM.disState);
13       absTran = extractAbstractTran(tree);
14       A-FSM.addSdis(SA-FSM.disStates, absTran);
15       A-FSM.addSadaptive(SA-FSM.adaptiveState, SA-FSM.adaptiveTran);
16    }
17    return A-FSM;
18 }
```

Fig. 2. Pseudo-code of the RINGA abstraction algorithm.

used for runtime verification. Therefore, to apply model checking to self-adaptive software, the proposed framework performs precomputing to abstract the SA-FSM in the form of the equations (i.e., one of the A-FSM transitions) so that the software can be analyzed quickly with low computing power at runtime. Abstracting the SA-FSM requires a considerable amount of computing power. However, by performing this precomputation, the proposed framework can analyze the system status simply by calculating the equations at runtime. The A-FSM definition and abstraction algorithm are described below.

A-FSM is a tuple (S, →, s$_0$, AP, L), where

- S is a set of states,
- S is classified into four types of subsets, $\{S_{start}, S_{sat}, S_{dis}, S_{adaptive}\} \subseteq S$,
- → is the set of transitions, and $\{\rightarrow_{A\text{-}FSM}, \rightarrow_{trigger}\} \subset \rightarrow$,
- $\rightarrow_{A\text{-}FSM} \subseteq \{S_{start} \times S_{dis}, S_{start} \times S_{sat}\}$ is the transition relationship,
- $\rightarrow_{trigger} \subseteq \{S_{dis} \times S_{adaptive}\}$,
- $s_0 \subseteq S_{start}$ is the initial state,
- AP is a set of atomic propositions, and
- L: $S \rightarrow 2^{AP}$ is a labeling function (2^{AP} denotes the power set of AP).

There are four types of states in A-FSM; in other words, S_{sat}, S_{dis}, and $S_{adaptive}$ are the same for A-FSM and SA-FSM. There is an initial state in A-FSM that is included in S_{start}. Every transition starts at s_0 and is directly connected to S_{sat} or S_{dis}. The A-FSM transitions indicate a path from s_0 to the connected S_{sat} or S_{dis} states based on the extracted SA-FSM, and such transitions are expressed in the form of equations that consist of SA-FSM transitions and mathematical operators. If A-FSM reaches a state in S_{sat}, a requirement has been satisfied. However, if A-FSM reaches S_{dis}, the software has failed to satisfy a requirement. Therefore, every S_{dis} state is connected to at least one $S_{adaptive}$ state, and if A-FSM reaches S_{dis}, the connected trigger performs the adaptive strategies associated with it.

A-FSM transitions are calculated at runtime in the proposed framework. If the A-FSM results show that there is a transition to reach S_{dis}, the self-adaptive software needs to adapt because reaching S_{dis} indicates that it is likely that some requirements have not been satisfied. Therefore, the purpose of extracting A-FSM from SA-FSM is to find every reachable path from S_0 to S_{sat} or S_{dis}. The purpose of this extraction is presented below using the intuitive semantics of temporal modalities [4].

Let m = (S, →, S$_0$, AP, L) be a SA-FSM, and ***Path***(*δ*) denote a path to satisfy temporal modalities *δ*. First,

$$\varphi_i = \bigcup Path(\exists \blacklozenge S_{sat}(i)) \tag{1}$$

Here, $S_{sat}(i)$ is the i-th state of S_{sat}. Notation ◇ indicates "eventually" (i.e., now or eventually in the future), and ∃ indicates that there exists at least one. Therefore, $\exists\Diamond S_{sat}(i)$ presents the existing path to reach the i-th state of S_{sat}, if such a path exists. In addition, *Path* $\{\exists\Diamond S_{sat}(i)\}$ denotes a path to reach the i-th state of S_{sat}. Finally, φ_i is a union of the existing paths from s_0 to $S_{sat}(i)$; thus, φ_i denotes the set of all reachable paths from s_0 to the i-th state of S_{sat}. Next,

$$\omega_j = \bigcup Path(\exists \blacklozenge S_{dis}(j)) \tag{2}$$

Here, $S_{dis}(j)$ is the j-th state of S_{dis}. Similar to Eq. (1), ω_j is a union of the existing paths from s_0 to the j-th state of S_{dis}; thus, ω_j denotes the set of all reachable paths from s_0 to S_{dis}.

The transition of A-FSM using φ and ω is given by

$$\rightarrow_{A-PSM} = \{\{\varphi_1 \dots \varphi_n\}, \{\omega_1, \dots, \omega_m\}\} \tag{3}$$

In Eq. (3), set S_{sat} contains *n* states, and set S_{dis} contains *m* states. In addition, $\rightarrow_{A\text{-}FSM}$ is a pair of φ_i and ω_j, and it indicates the set of all the reachable paths to reach a state that is an element of S_{sat} or S_{dis}. Finally, $\rightarrow_{A\text{-}FSM}$ represents the set of all reachable paths from s_0 to the states of S_{sat} and S_{dis}.

3.2.3. SA-FSM to A-FSM

In this section, we describe how to obtain A-FSM from SA-FSM. We revise the state elimination algorithm [25] to abstract SA-FSM in the form A-FSM. In brief, SA-FSM is changed into a tree data structure to extract the paths to reach S_{sat} and S_{dis}. After extracting these paths, the tree is transformed into equations (i.e., $\rightarrow_{A\text{-}FSM}$ of A-FSM). Fig. 2 lists the pseudo-code of the abstraction algorithm. The input of the abstraction algorithm is an SA-FSM and its output is an A-FSM. To abstract an SA-FSM into an A-FSM, two loops are needed (lines 6 to 10 and 11 to 16). These loops repeat for each S_{sat} and S_{dis} state (lines 6 and 11). In these loops, the A-FSM transitions are extracted from the input SA-FSM (lines 7 to 8 and 12 to 13). To extract A-FSM transitions, the paths to reach S_{sat} and S_{dis} are extracted as a tree structure (line 7 and 12). After extracting the tree structure, it is translated as an equation (lines 8 and 13). The extracted equation implies φ (Eqs. (1)) or ω ((2)) because both equations include all reachable paths to reach S_{sat} or S_{dis}. After extraction, the equation is added to the output A-FSM as transition (lines 9 and 14). This process implies Eq. (3), because the equation represents the set of all reachable paths to S_{sat} and S_{dis}. Unlike the first loop,

```
 1 // Input: State of SA-FSM
 2 // Output: Tree structure with paths to reach
 3 //            the state of SA-FSM (i.e., input state)
 4
 5 Tree extractPath(state){
 6   returnTree.root = state;
 7   while(state.hasIncomingTran){
 8     if(fromState == initialState)
 9       returnTree.addEndNode();
10     else if(isDuplicated(fromState))
11       returnTree.eliminateIterativeBranch();
12     else
13       returnTree.addChildNode();
14       state.addIncomingTran();
15   }
16   return returnTree;
17 }
```

Fig. 3. Pseudo-code of extracting path from SA-FSM.

$S_{adaptive}$ states are added to the output A-FSM in the second loop (line15) because the S_{dis} states are connected with $S_{adaptive}$ states (i.e., via $\rightarrow_{trigger}$) by the definition of A-FSM. After all loops, the output A-FSM is returned (line 17). Next, we describe the details of the algorithm to extract an A-FSM transition (lines 7 to 8 and 12 to 13).

Fig. 3 shows the pseudo-code of the SA-FSM path-extracting algorithm. This algorithm is one step of the A-FSM extraction (i.e., lines 7 and 12 in Fig. 2) and returns a tree structure that contains the paths to reach the input state. At the beginning of the algorithm, the input state is set as the root node of a tree structure (line 6), and the loop repeats until there is no connected incoming transition (lines 7 to 15). In other words, the loop terminates when all incoming states are initial states, duplicated states, or no state. In the loop, if the connected state is an initial state, it adds the initial state and its transition as a leaf node of the tree structure (lines 8 to 9). If the connected state is a duplicate state (e.g., the state is already added to the tree), the loop deletes one branch related to the duplicated state to prevent iterations, e.g., the loop deletes the duplicated branch (lines 10 to 11). In addition, if the connected state is not only an initial state, but also duplicated state (line 12), the loop adds a state and transition to the tree (line 13) and also adds the incoming transitions of the added state to extract the paths to reach the initial state (line 14). When adding a connected state in the loop, if there is single incoming state, it is added as a child node, and if there are multiple incoming states, they are added as siblings to that child. Therefore, sibling nodes indicate that there are multiple paths to reach the input state. After the loop, the tree contains reachable paths from the initial state to the input state, and the final tree is returned (line 16).

After the path extraction algorithm, the returned tree structure is transformed into an equation form (i.e., $\rightarrow_{A\text{-}FSM}$ of A-FSM) (lines 8 and 13 in Fig. 2). Fig. 4 shows the pseudo-code of the algorithm to transform the tree structure into the A-FSM transition. Input of this algorithm is a tree structure (i.e., lines 8 and 13 in Fig. 2), and the output is an A-FSM transition. The A-FSM transition algorithm consists of a loop that repeats until every node of the input tree is visited (lines 6 to 11). If a node has a sibling relationship, its transition is converted as an addition (+) or "or" (||) operation because the sibling relationship implies that there is another path to reach the state (lines 7 to 8). In addition, if a node has one node, its transition is converted into a multiplication (*) or "and" (&&) operation because, if one node cannot be reached, the remaining nodes are not reachable as well (lines 9 to 10). After the loop, an A-FSM transition that consists of a combination

```
 1 // Input: Tree structure
 2 // Output: A-FSM transition
 3
 4 absTran extractAbstractedTran(tree){
 5   absTran = null;
 6   while(tree.nextNode){
 7     if(tree.nowNode.hasSibling)
 8       absTran.ADDorOR(tree.nowNode);
 9     if(tree.nowNode.hasChlid)
10       absTran.MULorAND(tree.nowNode)
11   }
12   return absTran;
13 }
```

Fig. 4. Pseudo-code for the tree structure to A-FSM transitions transform algorithm.

of several transitions is generated, and the A-FSM transition is returned (line 12).

To better understand the extraction of an A-FSM transition, we describe the abstraction algorithm with a simple SA-FSM as depicted in Fig. 5. Here, the SA-FSM example consists of a satisfied state (state 5), a dissatisfied state (state 3), an adaptive state (state 6), and three normal states (states 0, 1, and 2). In addition, the SA-FSM contains an adaptive transition (transition g), and seven normal transitions.

As mentioned earlier, the A-FSM transitions indicate a path for reaching S_{sat} or S_{dis} states to verify whether the self-adaptive software satisfies or does not satisfy the requirements at runtime. We need a path to reach the S_{sat} or S_{dis} states of SA-FSM, and thus the abstraction algorithm starts at S_{sat} or S_{dis} and moves back to the initial state s_0. In Fig. 5, there are two abstraction starting points, states 3 and 5. In the SA-FSM example, we demonstrate abstraction algorithm for state 3. This starting point state is located at the root node of the tree structure (line 11 in Fig. 2). After setting the root node, the tree expands its children recursively (the algorithm in Fig. 3). A node can have children such that the transition is directly connected to SA-FSM. In the example, states 1 and 2 are the child node of node 3, and their branch values are d and c (i.e., line 12 in Fig. 3). However, if the edge is connected iteratively, it cannot be a child node because we want to extract paths to reach S_{sat} or S_{dis}, and iterations result in duplicate paths. Furthermore, considering such iterations could result in the state explosion problem. Therefore, if an iteration appears when expanding the tree, the exploration of the related states stops (line 10 in Fig. 3). Nevertheless, the tree structure expands when it reaches initial state s_0 (line 8

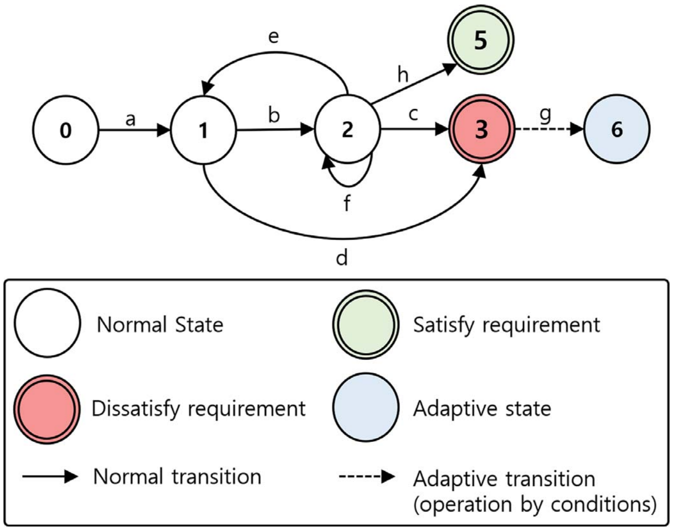


Fig. 5. Simple SA-FSM example.

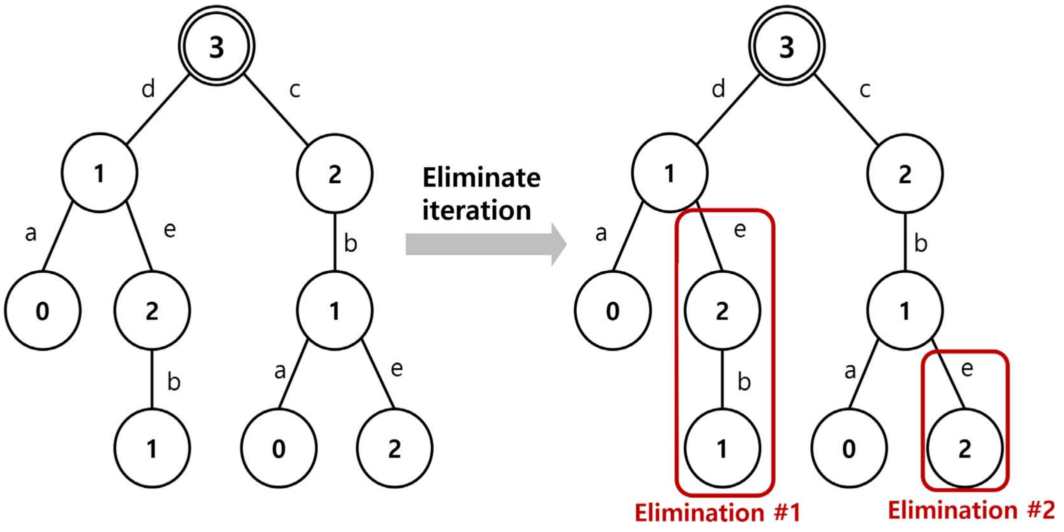


Fig. 6. Finite state machine transformed into a tree structure and elimination of its iterations.

in Fig. 3). The right side of Fig. 6 shows the tree structure transformed from the finite state machine example after the iterations have been eliminated. On left side of Fig. 6, leaf nodes 1 and 2 have a parent with the same name; therefore, these two nodes should be deleted.

After converting the SA-FSM into a tree structure, the translated tree can be converted into equations. This involves two steps (Fig. 4): first, the parent relationships are converted into an * or && operation (lines 9 to 10 in Fig. 4). Because the parent relationships indicate a path, if one edge (transition) is unable to reach the next state, the remaining nodes (states) cannot be reached, and thus, we use the * or && operation. In the second step, the sibling relationships are converted into an + or || operation because, even when one edge (transition) is unable to reach the next state in a sibling relationship, the other sibling edge (transition) may be able to reach it (lines 7 to 8 in Fig. 4). If the SA-FSM transitions are Boolean, an A-FSM transition is converted using && and ||. Otherwise, if the SA-FSM transitions are expressed as a number between 0 and 1, the A-FSM transition is converted using the + and * operators. Fig. 7 shows an example of the conversion of a tree into an A-FSM transition.

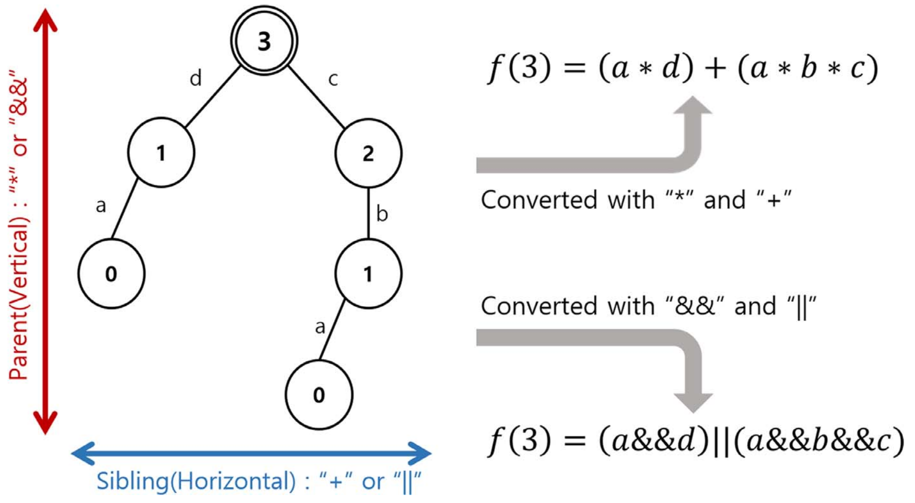
As shown in Fig. 7, there are two paths for reaching state 3: 0→1→3 and 0→1→2→3. There are several methods for determining the A-FSM transitions. Assume that the SA-FSM transitions are expressed as Boolean types. The A-FSM transition result is “true” when one of the two paths is true. When both paths are “false,” the A-FSM transition result is “false.” Therefore, if the A-FSM transition result is “true,” this implies that there is at least one path that can reach state 3. Otherwise, we assume that the SA-FSM transitions are expressed by integers 0 or 1. The A-FSM transition result has several possible ways to reach state 3. If both paths are true, the A-FSM transition result is 2. When only one path is true, the A-FSM transition result is 1. Naturally, if the result is 0, there is no path for reaching the state. Output can also be probability values between 0 and 1; they reflect the possibility of reaching a specific state. Moreover, the adaptive state (state 6) is attached to a dissatisfied state (state 3) by the A-FSM definition (line15 in Fig. 2). Finally, after applying the abstraction algorithm at state 5 (lines 6 to 10 in Fig. 2), an A-FSM can be extracted as Fig. 8.

We applied this abstraction method at the design-time part of the proposed framework. As mentioned in Section 3.2.2, the abstraction process requires a considerable amount of computing resources, but the design-time process reduces this requirement when the software is verified at runtime. Section 4.2 presents an evaluation of the proposed abstraction method and its runtime efficiency.

3.3. RINGA runtime process

In the previous section, we described the design-time process of the proposed framework for designing the SA-FSM and A-FSM. In addition, designed models are used to validate self-adaptive software in each MAPE loop at runtime. In this section, we describe how to validate self-adaptive software using the runtime verification results. The runtime

Fig. 7. Converting the tree into A-FSM transitions.


Parent(Vertical) : "*" or "&&"
Sibling(Horizontal) : "+" or "||"
$f(3) = (a * d) + (a * b * c)$
Converted with "*" and "+"
Converted with "&&" and "||"
$f(3) = (a\&\&d)||(a\&\&b\&\&c)$

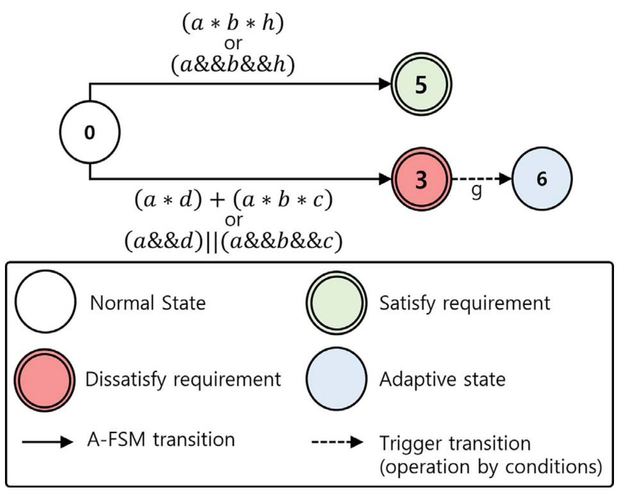


Fig. 8. Conversion of SA-FSM to A-FSM.

process for the proposed framework consists of a MAPE loop, and details are provided in the following sections.

3.3.1. Monitoring

The monitoring process is responsible for data collection from the environment and internal software changes. It also correlates the data. The SA-FSM transitions were defined for monitoring data and triggers when it was designed. Therefore, the monitoring process is responsible for checking SA-FSM transition values at runtime.

3.3.2. Analyzing

The analysis process analyzes the symptoms related to adaptation situations using the monitored data. The A-FSM transitions imply that an adaptation point is reached; they consist of a combination of SA-FSM transitions. Therefore, the analysis process calculates the A-FSM transitions, and deduces whether the software has reached and adaptation point. Furthermore, A-FSM consists of precalculated results for SA-FSM so it requires little computing power and completes quickly at runtime.

3.3.3. Planning

The planning process determines what is to be changed and how. This process triggers an adaptive strategy when adaptation is required. If the analysis process indicates that there is a reachable path to an adaptation point, the planning process selects the related triggers defined in SA-FSM. The selected triggers are then sent to the execution process.

3.3.4. Executing

The executing process is responsible for activating the adaptive strategies. Therefore, the planning process selects the triggers for adaptation to the environment, and the execution process implements them. As mentioned in the previous section, the proposed framework assumes that software should be implemented with the referenced SA-FSM, and thus a change in the transition values actually impacts the software. Therefore, the execution process can implement the software by changing the values of the trigger transitions.

4. Empirical evaluation

This section discusses a set of experiments for empirical evaluation of the proposed self-adaptive framework, RINGA. We implemented a prototype of RINGA using Java 1.8.0 to ensure compatibility with various devices. This section consists of the design-time performance and runtime performance tests. In RINGA, a designed finite state machine (SA-FSM) is generated as a runtime model (A-FSM) to reduce verification time and computation at runtime. Moreover, the design-time process is executed only once and does not impact at runtime process. Therefore, we divided the evaluation into design-time and runtime evaluation. The details are provided in the following sections.

4.1. Design-time performance

We randomly generated well-formed finite state machines for our experiments. All the states have at least one in-transition connection, with the exception of the initial state. The experimental data consists of different state sizes with two out-transitions and one end-state. We generated an experimental dataset to evaluate the worst case for the proposed framework. Every state in the test cases has two transitions, with the exception of the end state, which only has one in-transition. Therefore, every transition has to be explored to obtain an A-FSM from a SA-FSM.

The first experiment measures the time required to make A-FSMs from the experimental finite state machine data set. The data set consists of 20 states for each finite state machine, and we calculated the mean of each data set. We executed the proposed method in different hardware environments (two laptops, two desktops, one server and four different Android phones). Table 2 lists the conditions of the different hardware environments [26,27], i.e., CPU generation, number of cores, and memory size.

The results obtained for increasing number of states are shown in Fig. 9. More computation time is required when the number of states increases. Naturally, environments with high computing power require less calculation time. When the number of states is 45, only 9.3 s are required for low computing power (i.e., the Galaxy S5). Moreover, there is no difference between the environments when the number of states is less than 30, but differences in performance occur for more than 30. In addition, there are no significant differences between the high-power computing devices (i.e., laptops, desktops, and the server) except for the Intel® i7-2620 M laptop because this technology at least two years behind the other CPUs. However, the results show that high computing power is suitable for design-time. Nevertheless, the design-time phase is executed once before runtime and not until the finite state machine changes. Therefore, it does not impact the execution verification at runtime.

Furthermore, we performed similar experiments for different out-transition numbers. We fixed the state size to 15 and increased the transition size. We also generated the worst-case experimental data for performing the proposed framework, as indicated earlier in this section. The results of these experiments are shown in Fig. 10. Similar to the previous results (Fig. 9), more computational time is required the when the number of transitions increases. Moreover, high computing-power environments require less calculation time.

The results of the design-time experiments suggest that, if the finite state machine has several states and transitions and the running device has low computing power such that the design-time process cannot be calculated within a reasonable time, a high-computing power device should be used to calculate the A-FSM for low computing-power

Table 2
Details of the hardware environments for measuring performance.

Hardware	CPU Clock	CPU Core	RAM	Operating system
Laptop (Intel® i7-2620 M)	2.7 GHz	2	8 GB	Windows 7
Laptop (Intel® i5-5200 U)	2.2 GHz	2	8 GB	Windows 10
Desktop (Intel® i5-4670)	3.4 GHz	4	16 GB	Windows 10
Desktop (Intel® i5-6500)	3.2 GHz	4	8 GB	Windows 10
Server (Intel® Xeon® E3-1230 L v3)	1.8 GHz	4	4 GB	Windows 10
Samsung Galaxy A8	1.8 GHz	8	2 GB	Android 5.1.1
Samsung Galaxy S5	2.5 GHz	4	2 GB	Android 5.1.1
Samsung Galaxy S7	2.6 GHz	8	4 GB	Android 6.0.1
Samsung Galaxy Note 5	2.1 GHz	8	4 GB	Android 6.0.1

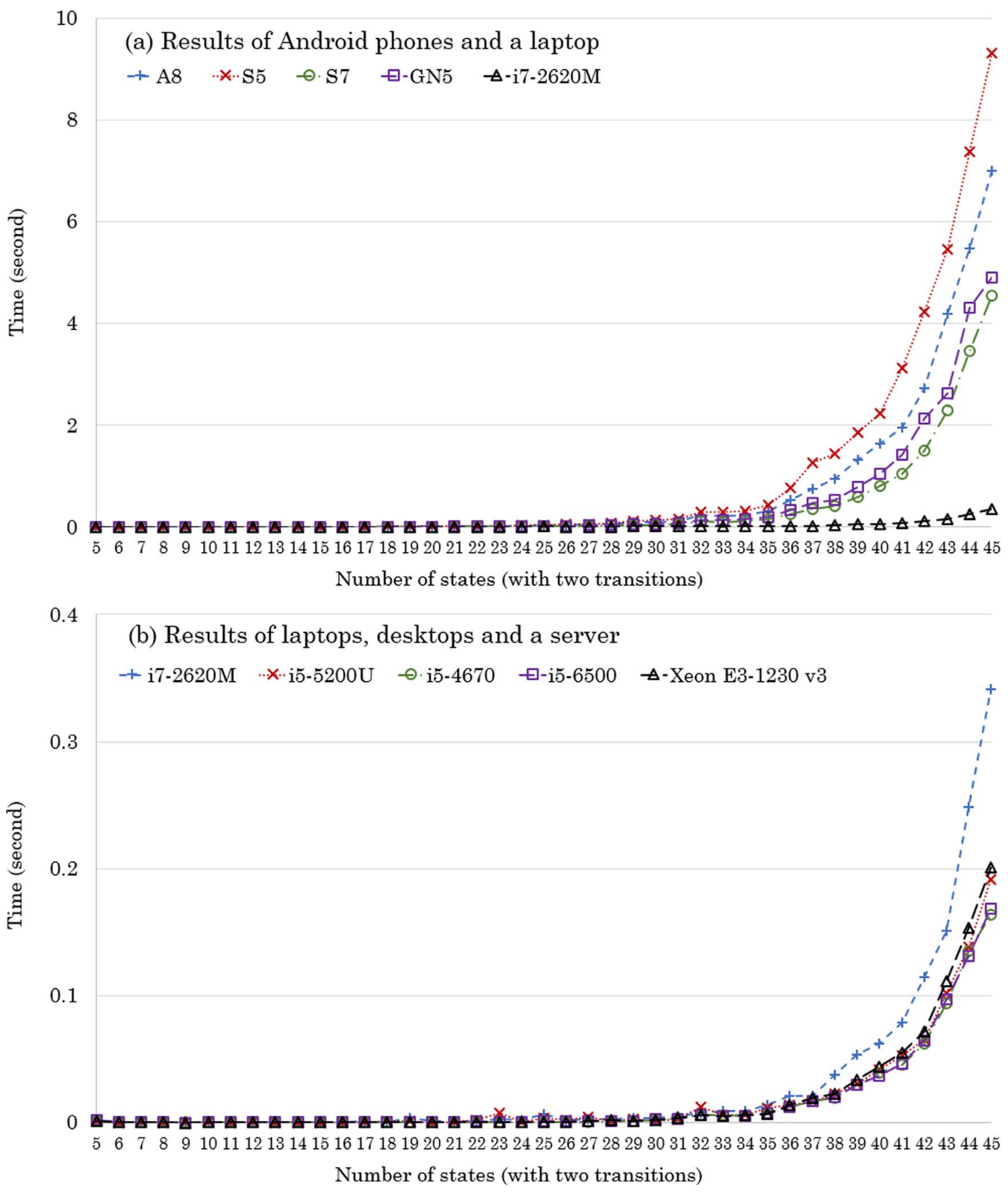


Fig. 9. Results of extracting A-FSMs with increasing number of states.

running device. In this way, the low computing-power running device (e.g., mobile or IoT devices) can operate the MAPE loop by calculating the A-FSM only at runtime.

4.2. Runtime performance

Self-adaptive software requires rapid verification at runtime to adapt to a changing environment, and thus verification time is an important consideration [2,3]. In this section, we perform experiments on runtime performance using the previous experimental data set. As we mentioned earlier, the experimental data (i.e., SA-FSMs) consist of different sizes of states or out transitions with one end node. Therefore, experimental data (i.e., A-FSMs) for runtime consist of an initial node, an end node, and an A-FSM transition (i.e., a transition with an equation form). Note that we consider various environment events (i.e., values of transitions) by random generation. Therefore, the results of the experiment denote average values of different environment events. However, different environment events do not have an impact on the result of runtime performance. For example, in the case of an SA-FSM that consists of 45 states and two transitions with 100 different environment events, the standard deviation is only 1.01 ms. First, we conducted the experiments on different devices and measured the time required to calculate the A-FSMs extracted in the previous experiments. Note that RINGA should consume little computing power in the runtime process, because it calculates only the equations (i.e., →A-FSM of A-FSM) at runtime. The results are shown in Figs. 11 and 12.

As the results show, more computation time is required when the state and transition sizes increase because, as the state and transition sizes of the finite state machine increase, the length of an A-FSM transition also increases. Naturally, high computing power requires less

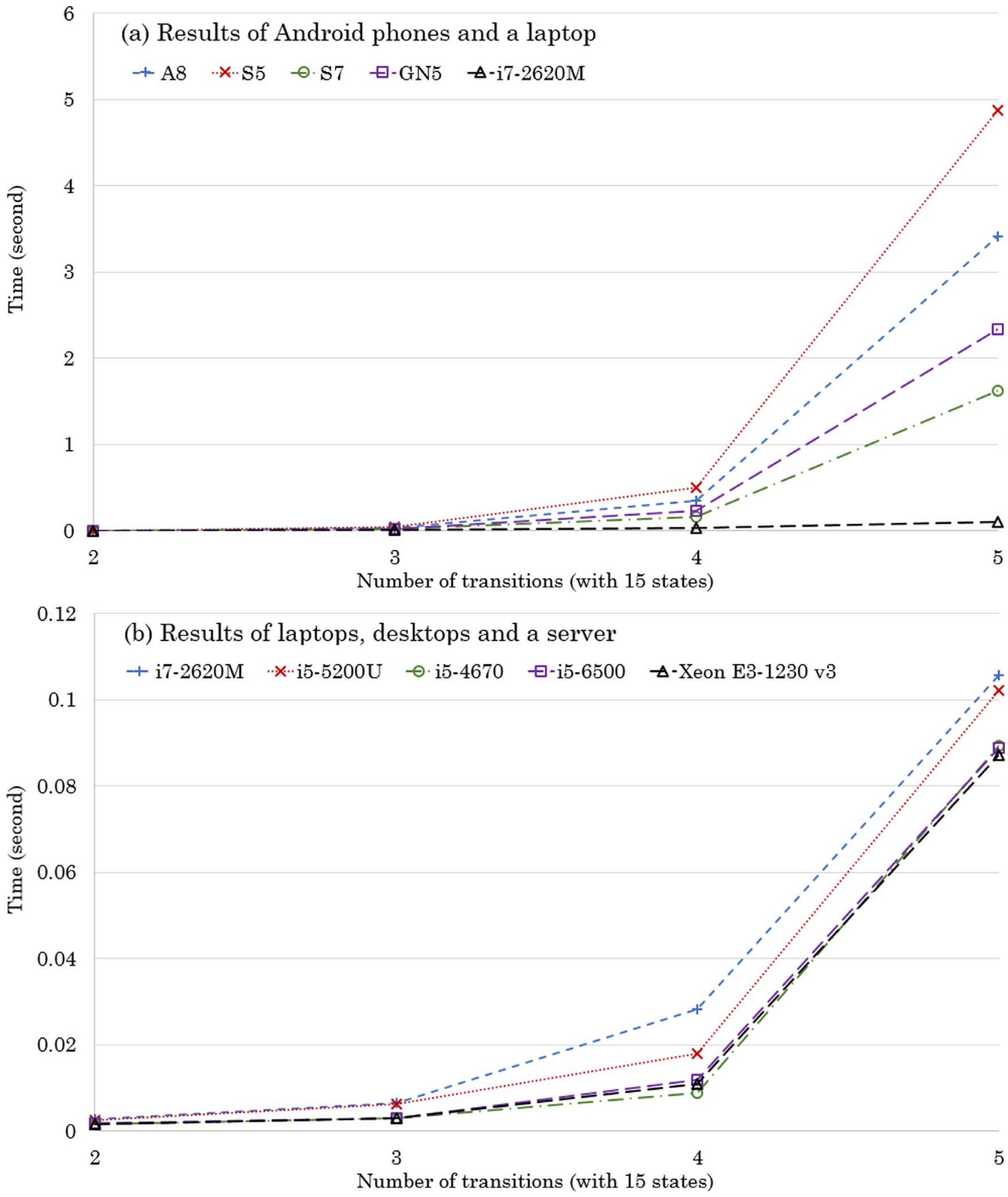


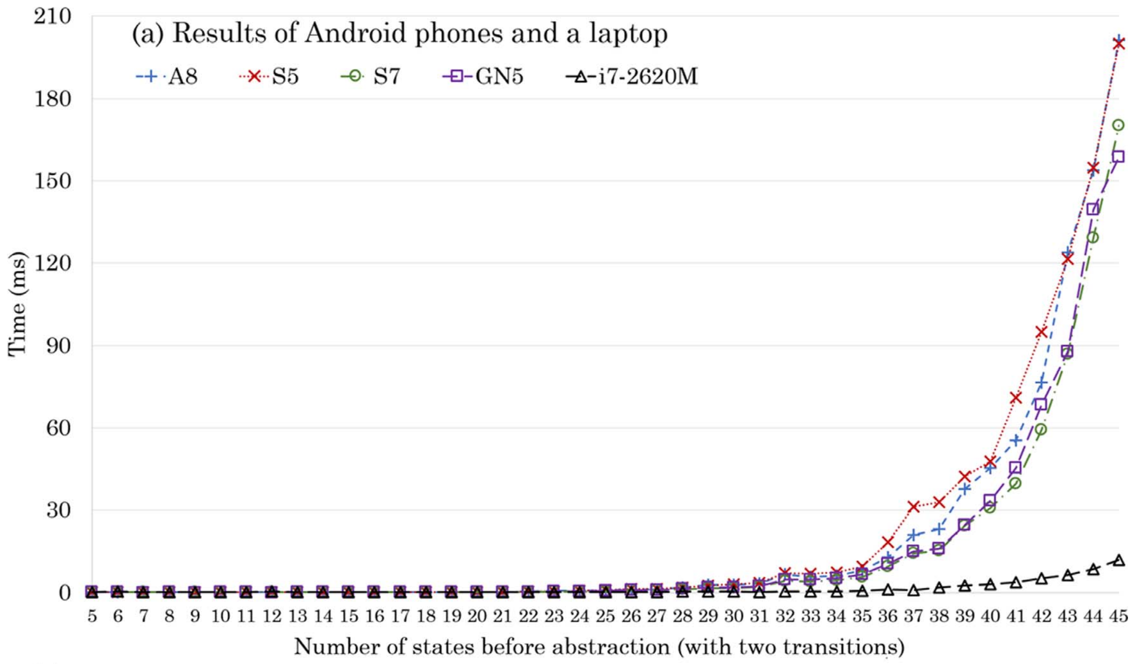
Fig. 10. Results of extracting A-FSM with increasing number of transitions (15 states).

time. However, all the devices calculate A-FSMs within a reasonable time. Even low computing-power devices (e.g., Galaxy S5) require less than 202 ms with 45 states and 90 transitions. These experiments suggest that the runtime of the proposed framework is suitable for low computing devices such as mobile phones after the design-time process is complete.

To demonstrate the performance of the runtime process of the proposed framework, we compared the proposed method with two model checking tools that use a finite state machine to express the system model: NuSMV [28–31] and CadenceSMV [32]. These tools, which are powerful tools used in the model verification area, are symbolic model checkers, and they are provided as open source. NuSMV and CadenceSMV are limited to Linux and Windows operating systems, and thus, we performed the experiments using the Intel® i7-2620 M laptop. We obtained A-FSMs that contain paths from the start to end state by performing a design-time process. Therefore, we measured the time required to achieve reachability from the start to end state using NuSMV and CadenceSMV. Using the intuitive semantics of temporal modalities [4], reachability is

$$\sigma = \exists \blacklozenge \mathbf{endstate} \tag{4}$$

where σ indicates that there is a path that eventually reaches the end state. To calculate σ, NuSMV and CadenceSMV obtain a path from the start to end states. The results of the runtime performances are compared in Figs. 13 and 14. As the results indicate, the proposed method is better than the path finding (i.e., reachability) of both NuSMV and Cadence SMV. These tools differ from the proposed method when the state and transition sizes increase. NuSMV and CadenceSMV experience a monotonic change with increasing states and transitions because they terminate model checking when they find a path that reaches a specific



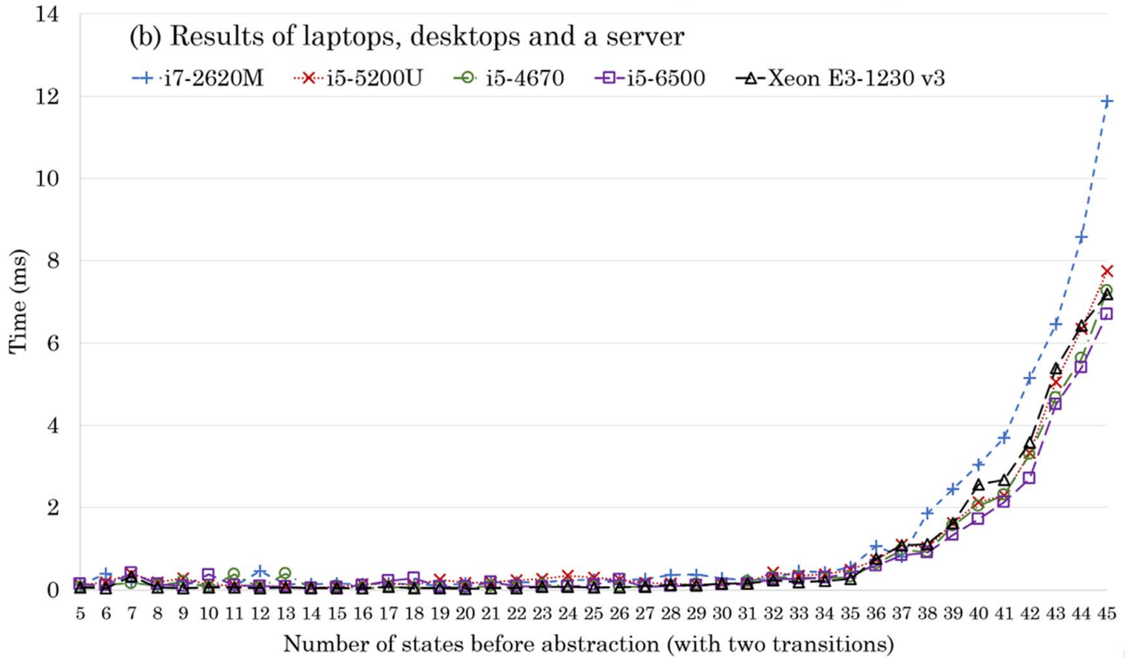


Fig. 11. Results of calculating A-FSM with increasing number of states.

state, as shown in Eq. (4). Therefore, NuSMV and CadenceSMV only provide a single path for reaching the end state, but the proposed method can consider several cases for reaching the end state. In addition, the proposed method is faster than the other tools. Naturally, the proposed method has a precomputation process for converting the SA-FSM into an A-FSM. Nevertheless, the proposed method saves time at runtime when considering several model-checking cases.

5. Proof of concept: self-adaptive light control

In this section, we describe an example application to help understand the proposed method: self-adaptive light control. The example uses a simple IoT concept, and thus there two devices: a smartphone that contains the user requirements, control connections, and proposed MAPE loop, and an Arduino-based light control device with a wireless network module. The implemented application and device were simulated using six scenarios. The results of the scenarios show that the proposed RINGA framework can be applied at runtime. In Section 5.1, we provide the details of the proposed example application and its scenarios. In Section 5.2, we explain how the application and light control device were implemented. In Section 5.3, we show the simulation results of the six scenarios.

5.1. *Example application: self-adaptive light control*

5.1.1. *Overview of example application*

In this simple example application, there is a lamp with an illumination sensor, wireless communication module, and adjustable brightness controller. A user can set the lamp brightness through the mobile device. When the mobile device is connected to the lamp, it checks the user brightness requirements. Subsequently, the mobile device collects data from the environment and internal software changes, for example,

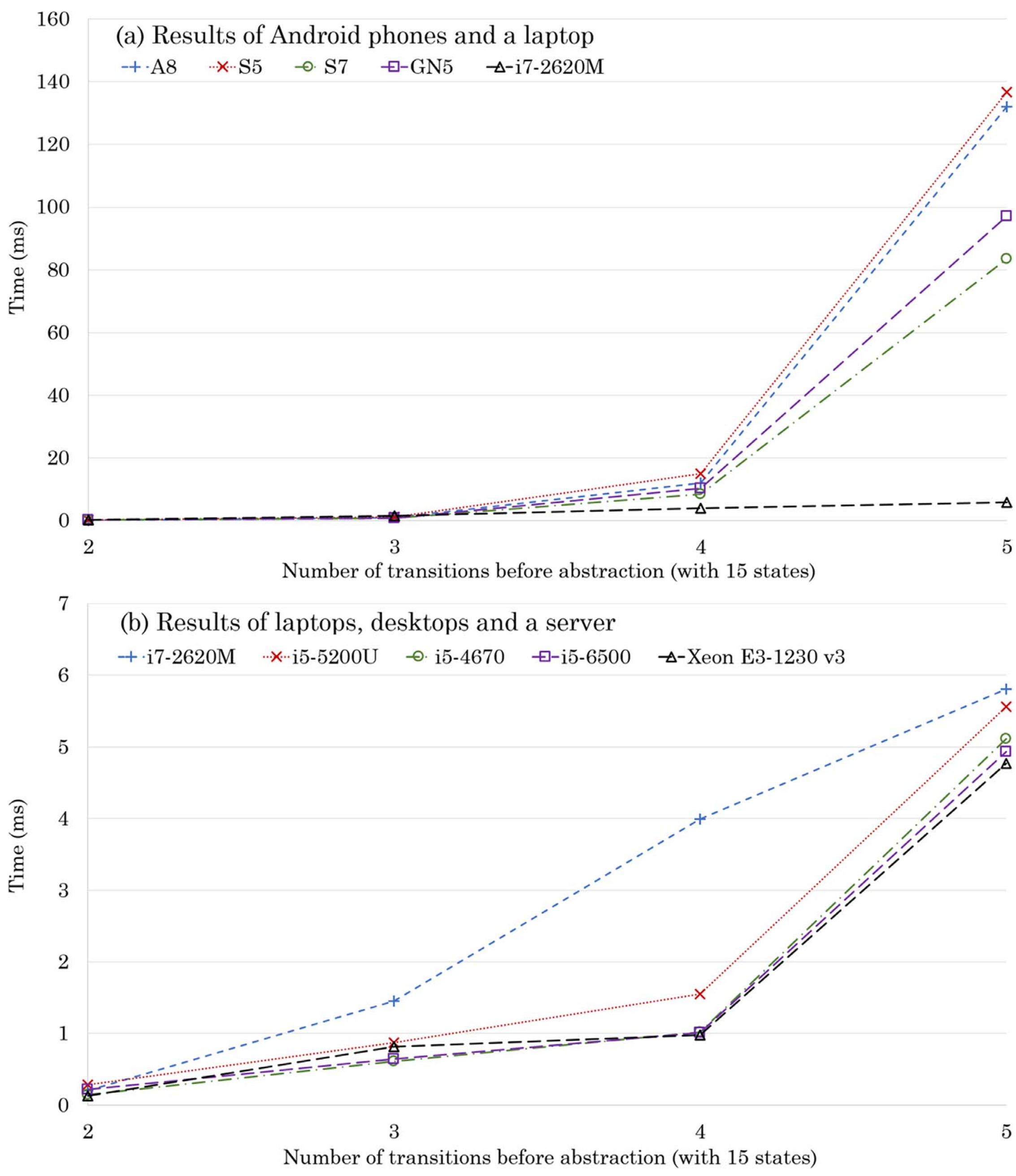


Fig. 12. Results of calculating A-FSMs with increasing numbers of transitions.

the illumination intensity, available sensor list, and requirements. After monitoring, the mobile device analyzes the monitored data and evaluates the adaptive strategies needed to satisfy the user requirements. If there are strategies that satisfy the requirements, the mobile device controls the related devices. However, if the mobile device loses its connection or requires more data, it attempts to reconnect or checks the monitoring data again. Fig. 15 shows an overview of the proposed example application.

In the example application, we assume that the user does not want sudden illumination changes, so the illumination should be changed gradually. However, there are two points for checking for adaptability: satisfaction of the user requirement and robustness to broken sensors. Based on these two checkpoints, we created six scenarios to measure the adaptability of the proposed example application:

- Scenario #1: The light controller adapts the external light to satisfy the user's requirement and starts with no connection. The external lights are fixed and do not change.
- Scenario #2: The light controller adapts the external light to satisfy the user's requirement. The external light suddenly becomes darker.
- Scenario #3: The light controller adapts the external light to satisfy the user's requirement. The external light suddenly becomes brighter.
- Scenario #4: The light controller adapts the external light to satisfy the user's requirement, and the illumination sensor is suddenly inoperative during the adaptation process. The external light is fixed and does not change.
- Scenario #5: The light controller adapts the external light to satisfy the user's requirement with a broken illumination sensor. The external light suddenly becomes brighter.
- Scenario #6: The light controller adapts the external light to satisfy the user's requirement with a broken illumination sensor. The external light suddenly becomes darker.

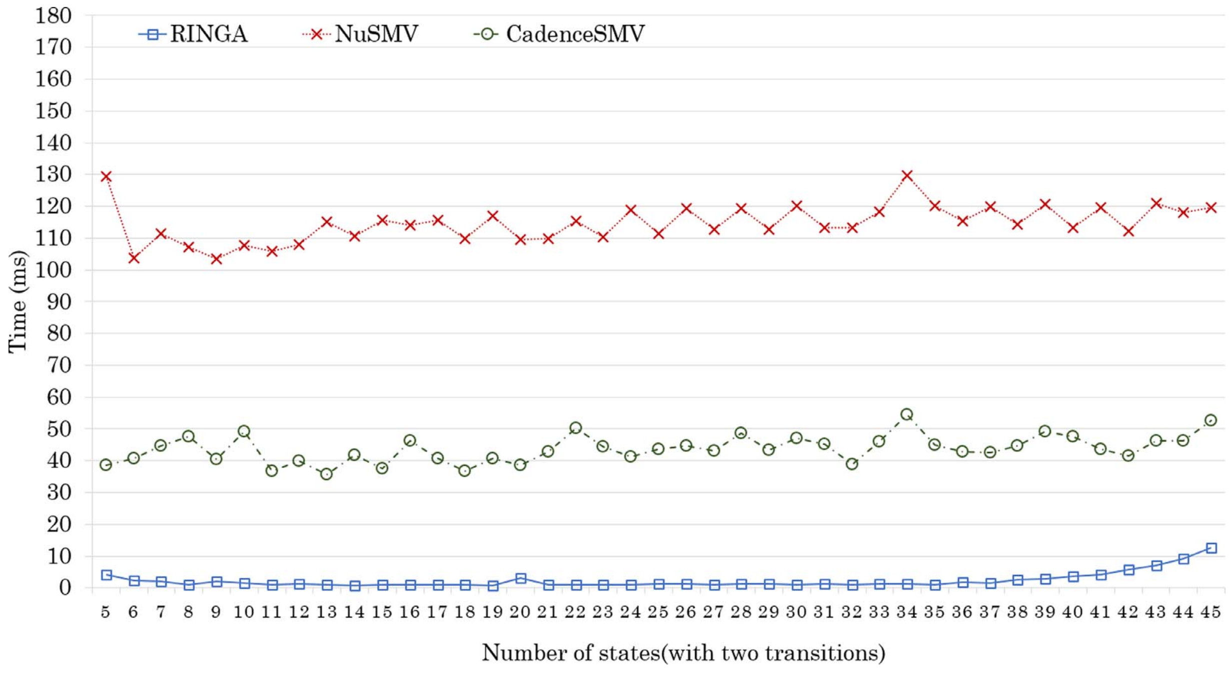


Fig. 13. Results of runtime comparison with increasing numbers of states.

Based on the example application and scenarios, we design the SA-FSM and A-FSM in Section 5.1.2. Moreover, we implement the application and light control devices in Section 5.2.1. The results of the scenarios are described in Section 5.2.2.

5.1.2. *Modeling of finite state machine*

To simulate the proposed framework, we design an SA-FSM based on the example application described in Section 5.1.1. Before designing the SA-FSM, we build the rule-based finite state machine as a plan. We use the term “plain finite state machine” to refer to the plan for a finite state machine for building the SA-FSM. Therefore, a plain finite state machine is intuitive and simple; it consists of only two state types (i.e., normal and end state) and directed transitions. We design a finite state machine that contains a single loop for the example application. Before designing the finite state machine, we classify necessary conditions and user requirements for satisfying the example application. The necessary conditions represent the operational conditions under which the application can operate normally. In addition, the requirements indicate that the application satisfies the requirements of the user. A plain finite state machine is designed with the necessary conditions and user requirements. Fig. 16 shows the designed plain finite state machine with 23 states and 29 transitions. The necessary conditions and related states are listed below:

- Necessary condition #1: a wireless network connection between the devices and power check (states S0 to S5, S7, and S8)

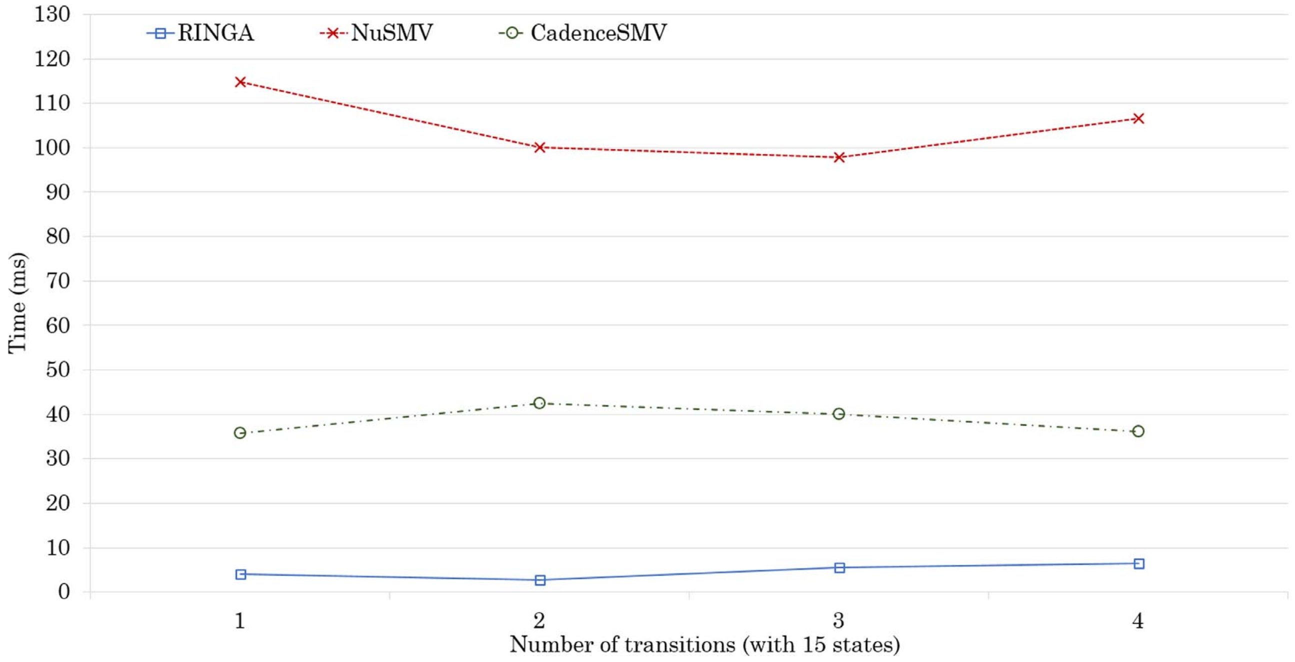


Fig. 14. Results of runtime comparison with increasing numbers of transitions.

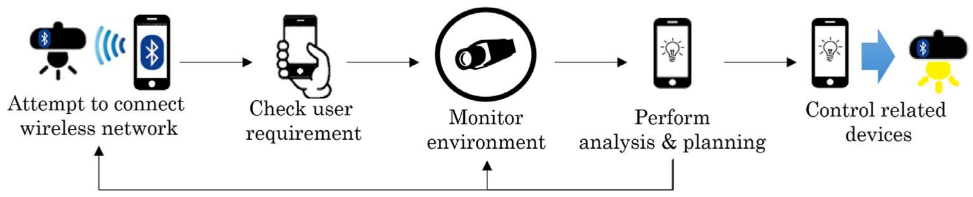


Fig. 15. Overview of self-adaptive light control example application.

- Necessary condition #2: the sensor value is readable (states S9 to S12)
- Necessary condition #3: the light is controllable (states S14 to S19)

The user's requirement and related states are below:

- Requirement #1: the application is shut down when the user wants to end it (states S5 and 6)
- Requirement #2: the user's desired light intensity is satisfied (states S13 to S20)

In this finite state machine, we omit details of the operation to prevent it from becoming too complex. For example, we omit the details of the reconnection process in states S3 and S4. The finite state machine has states for potential problems: losing a connection (state S2), a broken sensor (state S10), and brightness control (states S15 and S17). These three states are connected to the S22 (error) state. In addition, there are three end states: S6 and S20 are the states that end with a satisfied requirement, and S22 is the end state in which the requirement is not satisfied. The other states have their own operation, and the details are shown in Fig. 16. Note that building the plain finite state machine is not an essential process in RINGA. We build the plain finite state model to better understand the SA-FSM design (e.g., as a plan), and it is not needed if SA-FSM is designed immediately. We also want to show the possibility that a rule-based finite state machine can be transformed into an SA-FSM.

We design an SA-FSM from the plain finite state machine based on the definition in Section 3.2.1. First, we divide the states into four types (normal, satisfied, dissatisfied, and adaptive).

A *satisfied state* is an end state set where the software requirements are satisfied. Therefore, if the software reaches this state, it terminates normally; therefore, we select S6 (shutdown) and S20 (satisfy requirement). Because, S6 satisfies requirement #1 (e.g., user shutdown), and S20 satisfies requirement #2 (e.g., light intensity satisfied). After selecting a satisfied state, we change these states to end states. A *dissatisfied state* is an end state set in which the software requirements are not satisfied. Therefore, if the software reaches this state, it terminates abnormally. This type of state indicates that the software needs to be adapted, and thus after this state, an adaptation trigger is required. Therefore, we chose states that do not satisfy the user requirements and necessary conditions of the application. States S2 (connection lost) and S10 (disable sensor) violate necessary conditions #1 and #2. Further, S15 (natural light) and S18 (poor resource) may possibly violate requirement #2 and necessary condition #3. Therefore, these states containing potential failures are classified as dissatisfied states as they change the end states (S2, S10, S15, and S18). An *adaptive state* is a state in which adaptive activity is performed. If the software reaches this state, the related adaptive strategy is triggered. A *dissatisfied state* must connect to an adaptation trigger by definition. Therefore, states that are connected to dissatisfied states and contain adaptation triggers are selected as *adaptive states*. State S3 (attempt to check) is connected to S2 and is an adaptive strategy for condition #1. State S11 (check available sensor) is connected to S10 and is an adaptive strategy for condition #2. State S16 (reduce light) and S19 (increase light) are connected to S15 and S18, and these are adaptive strategies of requirement #2. Therefore, S3, S11, S16, and S19 are adaptive states. Before selecting the *normal states*, we delete states that lose connections, thus the S22 (error) state is deleted. State S22 was an end state to notify the user of an error, but connected states are translated as end nodes (*dissatisfied states*). Thus, that state loses all connections to reach itself and is deleted. A *normal state* is a state that does not impact software adaptation. Therefore, the remaining states are determined to be *normal states* (S0, S1, S4, S5, S7, S9, S12, S13, S14, and S17).

After selecting the states types, we remove and adjust the transition types. Transitions e3, e13, e20, and e25 are removed. Because S2, S10, S15, and S18 are both end states and *dissatisfied states*, there is no reason to connect them to S22. Moreover, some transitions are changed to *adaptive transitions*. The *adaptive transition* connects *dissatisfied states* and *adaptive states* by definition. Therefore, these transitions become adaptation triggers (e2, e14, e21, and e26). The other transitions are labeled as *normal transition*. Fig. 17 shows the SA-FSM design from the plain finite state machine for the light control application.

The *normal, satisfied requirement, dissatisfied requirement,* and *adaptive states* are represented by white, green, red, and blue circles, respectively. A *dissatisfied state* is connected to the *adaptive state*. The *adaptive state* is not reached when the *adaptive transition* is not operable. A *normal transition* is represented by a solid arrow, and an adaptive transition is represented by a dotted arrow.

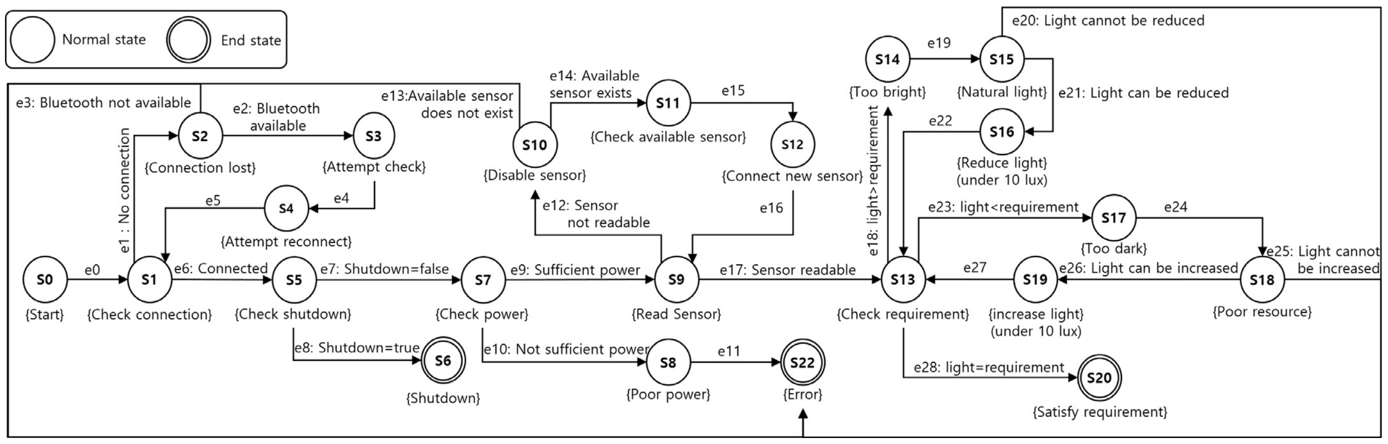


Fig. 16. Plain finite state machine for light control scenario.

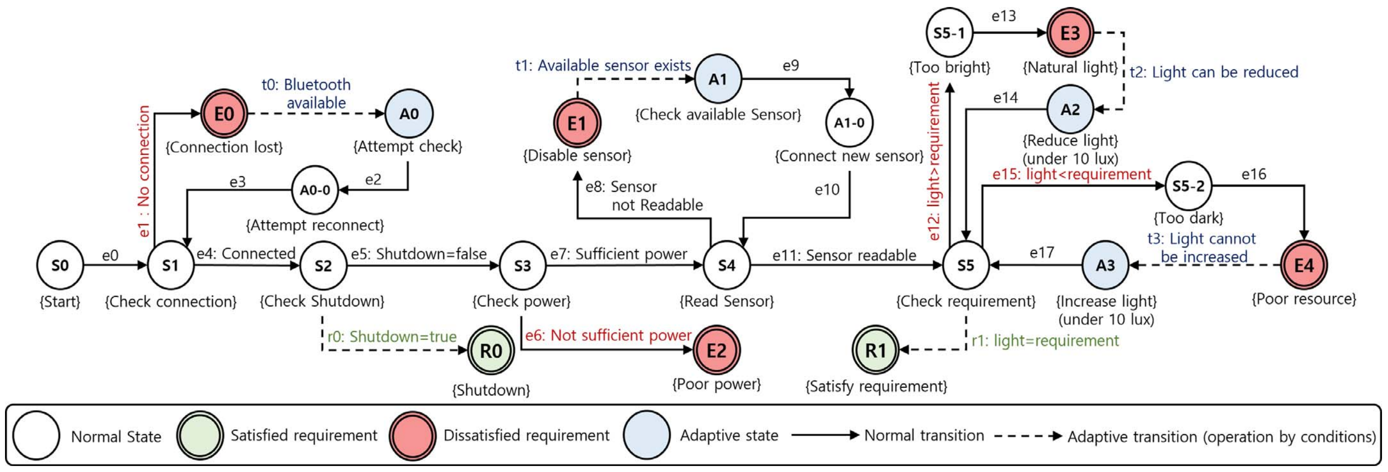


Fig. 17. SA-FSM for light control example application.

Table 3 lists the data-related transitions and triggers to be monitored.

After determining the finite state machine and monitoring data, we extract the A-FSM from the SA-FSM, as depicted in Fig. 18. The A-FSM defines an initial state as a connected *satisfied state* and a *dissatisfied state* as an A-FSM transition. Therefore, the A-FSM transition is extracted in the form of equations and connects related states. The initial state connects satisfied states (R0, and R1) and dissatisfied states (E0, E1, E2 and E4). Further, each dissatisfied state is connected to at least one adaptive state (trigger transition).

5.2. Implementation

To show the efficacy of the proposed framework at runtime, we implemented an Arduino-based lamp that uses Bluetooth communication to control the lamp's brightness (Section 5.2.1). We also implemented an Android application to control the self-adaptive lamp using the proposed framework (Section 5.2.2).

5.2.1. Arduino implementation

The pseudo-code for the Arduino-based lamp with simple functions is listed in Fig. 19. First, the lamp connects to the host device (line 7). In the example application, this is an Android phone. After connection, the lamp senses the illumination intensity (line 11), and then it sends this value to the host device (line 12). Then, the lamp receives an output value from the host device to adjust the illumination (line 15). Finally, the lamp adjusts its illumination based on the output value from the host device (line 16). This process continues every 100 ms until the host device sends a shutdown message (lines 9 and 19). The reason for the 100 ms delay (line 22) is that an Arduino board (Bluno board 2.0) cannot handle data if the host device sends data too quickly.

Table 4 lists the hardware components required for implementing the lamp, and Fig. 20(a) shows a deployment of the components using the Fritzing tool [33]. We used a Bluno board 2.0, which is a microcontroller board based on the Arduino Uno [34,35] and is integrated with a Bluetooth 4.0 module. (Fig. 20(a) does not show the Bluetooth module.) We deploy two illumination sensors to detect broken sensors. If two sensor values are significantly different, the host device determines that one of the sensors is broken. We use LED strips that consist of 45 LED lights to adjust the illumination. An LCD panel is deployed to show the lamp status. The brightness of the LED strips and LCD are amplified by a transistor. We also use a button with a 10-kΩ resistor to force termination. Fig. 20(b) shows a prototype of the lamp.

Table 3
Monitoring data and triggers.

Type	Data to be monitored	Related transitions
Monitoring data	Success or failure of the Bluetooth connection	e1, e4
	User shutdown command	e5, r0
	Sufficient power condition	e6, e7
	Check sensor operation	e8, e11
	Illumination value	r1, e12, e15
Triggers	Availability of Bluetooth network	t0
	Check assistance sensor	t1
	Light output value	t2, t3

5.2.2. Android application

We implemented an Android 5.1.1 application for the proposed design, which consists of the proposed MAPE loop. Fig. 21 shows the pseudo-code for this application. The entire process consists of a design-time process (lines 4 to 5) and MAPE loop (lines 7 to 20). The monitoring data and triggers are first mapped to an SA-FSM (line 4), which is abstracted to A-FSMs using the method described in Section 3.2 (line 5), and then the MAPE loop starts (line 7). During the monitoring process, the application collects monitoring data values and triggers (line 9). During the analysis process, the application calculates equations (i.e., $\rightarrow_{\text{A-FSM}}$ of A-FSM) extracted during design-time (line 12), and then the trigger is selected during the planning process using the results of the analysis process (line15). During the execution process, the application executes the triggers and displays its status. Then, the MAPE loop continues until the user shutdown message is received (line 7). Note that the lamp application processes data in 100 ms intervals, but the Android application process the MAPE-loop without delay. In addition, as mentioned in Section 5.1.1, we assume that the user does not want sudden illumination changes; hence, we change the illumination gradually. Although it depends on the environment, the illumination change is approximately under 10 lx.

We map the functions to the transitions in Table 3. If the trigger changes a transition value, the related function runs. For example, if the trigger indicates that value t2 is “true,” the application runs the function that sends a light-reduction message to the lamp. The application has a single activity view, and Fig. 22 shows a runtime screenshot, where the current illumination value appears at the top. The Bluetooth connection can be checked on the left button, and the shutdown button is on the right. The user requirement is entered between the connection check and shutdown buttons. In this screenshot, the user requirement is set to level 4 (400–600 lx). The “Now status” can be checked in the red-colored box. The bottom of the application displays the log data for the sensor value, previous lux, light output value, time, and A-FSM transition results.

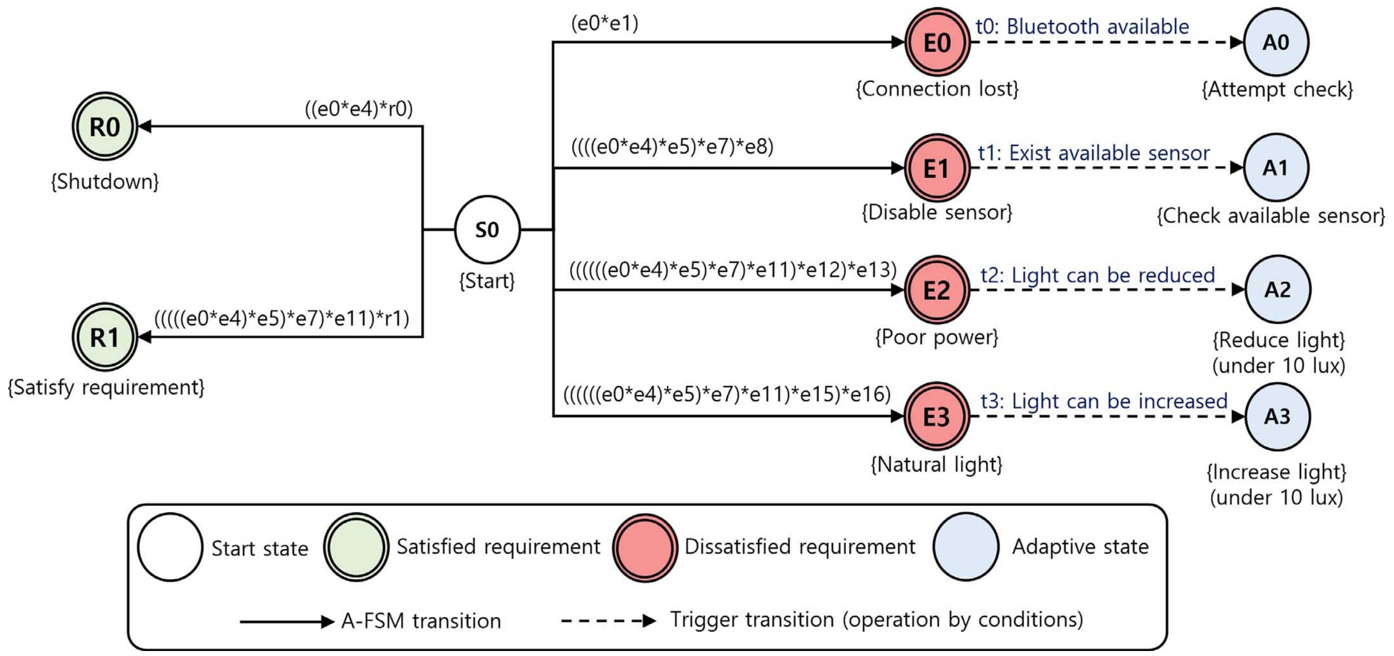


Fig. 18. A-FSM for the light control example application.

```
// Input : Output value to adjust illumination
//           Device shutdown message
// Output : Intensity of illumination value
class lightLamp
{
  // Connect with host device
  connectHostDevice();

  while(isShutDown){
    // Read illumination sensors and send to host device
    sensedVal  = readLuxSensor();
    sendIllumination(sensedVal);

    // Receive illumination value and adjust illumination intensity
    outputVal = receiveLux();
    setVal(outputVal);

    // Receive shutdown message
    isShutDown = receiveShutdown();

    // Wait 100 ms
    sleep(100);
  }
}
```

Fig. 19. Pseudo-code for the lamp.

Table 4
List of lamp components.

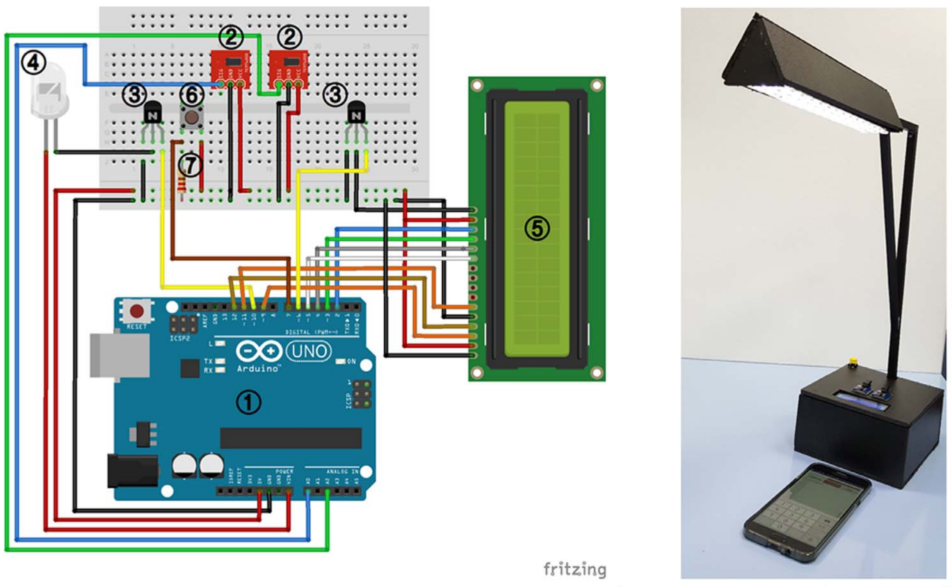
Number	Component name	Description
1	Bluno board 2.0 [35]	Microcontroller board with Bluetooth 4.0 module
2	Illumination sensor	Senses the intensity of illumination
3	Transistor	Amplifies the electrical power to adjust LED and LCD brightness
4	LED strips	Brightens the lamp
5	LCD module	Shows lamp status
6	Button	Forces termination
7	Resistor	Provides resistance

5.3. Results of the example application

We performed a simulation with the Android application and Arduino lamp based on the defined scenarios. Figs. 23–27 show the results for each scenario, where we set the user-required illumination intensity to a range between 220 and 280 lx. The blue area indicates that the user requirement has been satisfied and the system is operating normally. The red area indicates that the user requirement is not satisfied, and thus, the system needs to adapt to changes in the environment. Every result shows the environmental light levels and adapted light levels, which are controlled by the self-adaptive lamp. These illumination intensity results infer that the proposed method adapts to changes in the environment. As mentioned earlier, we assume that the user does not want sudden illumination changes, so the illumination is increased or decreased in steps of 10 lx when an adaptation is needed (Sections 5.1.1 and 5.2.2). Therefore, it appears that all the scenarios take time to satisfy a user's illumination requirement when in the dissatisfied states (E3 and E4). However, the application is continually executing adaptation actions (A2 or A3) in states E3 or E4.

Fig. 23 shows the results of scenario #1. This scenario requires adaptation to a monotonic environment change and Bluetooth connection. Therefore, the external light is fixed from 30 to 40 lx, and the application starts with no connection. In the first stage (0–1.3 s), the application calculates the equations (i.e., $\rightarrow_{\text{A-FSM}}$ of A-FSM in Fig. 18), determines that the system state is E0 (connection lost), checks t0 (Bluetooth available), and attempts to reconnect to the lamp. After connection is achieved in the second stage (1.4–3.9 s), because a user requirement was not inputted, the application determines that the system state is R0 (shutdown) from the results of A-FSM. After the user requirement is input (4–8.3 s), the application determines that the system state is E4 (poor resource), and the application checks t1 (light can be increased). In this situation, the application continually triggers A3 (increase light) until the user requirement is satisfied. In the last stage (8.4–11.8 s), the user illumination requirement is satisfied, and the A-FSM result shows that the system state is R1 (satisfied requirement).

Fig. 24 shows the results of scenario #2. This scenario is needed for adapting to a sudden decrease in external illumination. As shown in Fig. 24, the environment's illumination suddenly decreases from 260 to 60 lx over 1.3 s. Therefore, the application goes from state R1 (satisfied

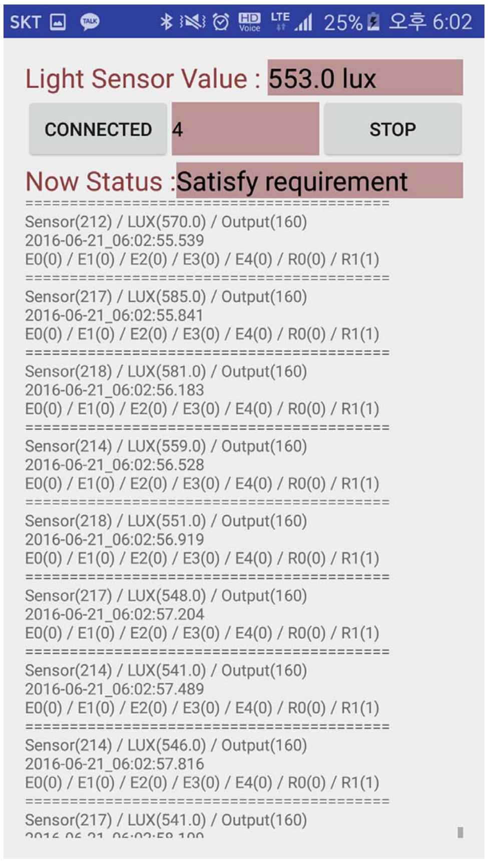


(a) Components for lamp deployment (b) Lamp prototype for proposed design

Fig. 20. Components for lamp deployment and lamp prototype for proposed design.

```
class LightControlApp
{
   // Abstracts FSM to A-FSM
   arrParameter[] = mapParameter(initFSM);
   arrAFSM[] = FSMtoAFSM(initFSM);

   while(isShutDown){
      // Monitoring
      readParameters(arrParameter[]);

      // Analysis
      resultCalAFSM[] = calculateAFSM(arrAFSM);

      // Plan
      arrTrigger[] = selectStrategy(resultCalAFSM[]);

      // Execute
      triggerExecute(arrTrigger[]);
      showStatus();
   }
}
```

Fig. 21. Pseudocode of light control application.



```
Light Sensor Value : 553.0 lux
CONNECTED 4 STOP
Now Status :Satisfy requirement
========================================
Sensor(212) / LUX(570.0) / Output(160)
2016-06-21_06:02:55.539
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(217) / LUX(585.0) / Output(160)
2016-06-21_06:02:55.841
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(218) / LUX(581.0) / Output(160)
2016-06-21_06:02:56.183
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(214) / LUX(559.0) / Output(160)
2016-06-21_06:02:56.528
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(218) / LUX(551.0) / Output(160)
2016-06-21_06:02:56.919
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(217) / LUX(548.0) / Output(160)
2016-06-21_06:02:57.204
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(214) / LUX(541.0) / Output(160)
2016-06-21_06:02:57.489
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(214) / LUX(546.0) / Output(160)
2016-06-21_06:02:57.816
E0(0) / E1(0) / E2(0) / E3(0) / E4(0) / R0(0) / R1(1)
========================================
Sensor(217) / LUX(541.0) / Output(160)
```

Fig. 22. Screenshot of the application at runtime.

requirement) to E4 (poor resource) from the results of A-FSM. Furthermore, the application checks t3 (light can be increased) and triggers A3 (increase light). The application operates adaptation state A3 until the user requirement is satisfied.

Fig. 25 shows the results of scenario #3. This scenario is needed for adapting to a sudden increase in the external illumination. As shown in Fig. 25, the environment's illumination suddenly increases from 64 to 271 lx over 1.2 s. Therefore, the application goes from state R1 (satisfied requirement) to E3 (natural light) from the results of A-FSM. Furthermore, the application checks t2 (light can be reduced) and triggers A2 (reduce light) continually until the user requirement is satisfied. The application operates adaptation state A2 until the user requirement is satisfied (after 3.6 s).

In scenario #4, the application needs to adapt to a monotonic illumination change and broken illumination sensor. Fig. 26 shows that the lamp sensor does not work at 2.3 s, and thus, the application determines that the system state is E1 (disabled sensor) by result of A-FSM. Subsequently, the application checks t1 (available sensor exists). In this case, the smartphone (Galaxy A8) has an illumination sensor, and therefore the application triggers the adaptive states A1 (check available sensor) and A1–0 (connect new sensor). The disabled illumination sensor is not revealed after 2.3 s, and thus the results after 2.3 s are based on the secondary illumination sensor built in the phone. As shown in Fig. 26, the application performs an adaptation process from 0

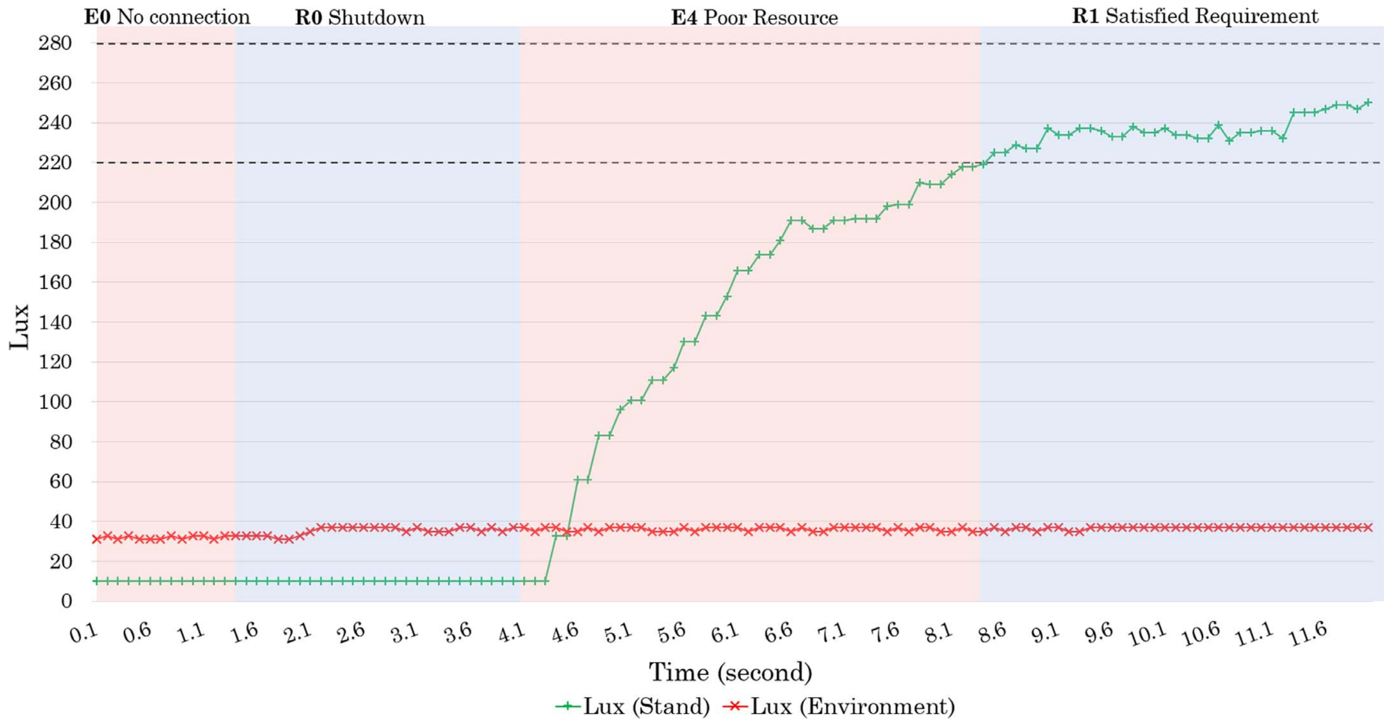


Fig. 23. Results of scenario #1.

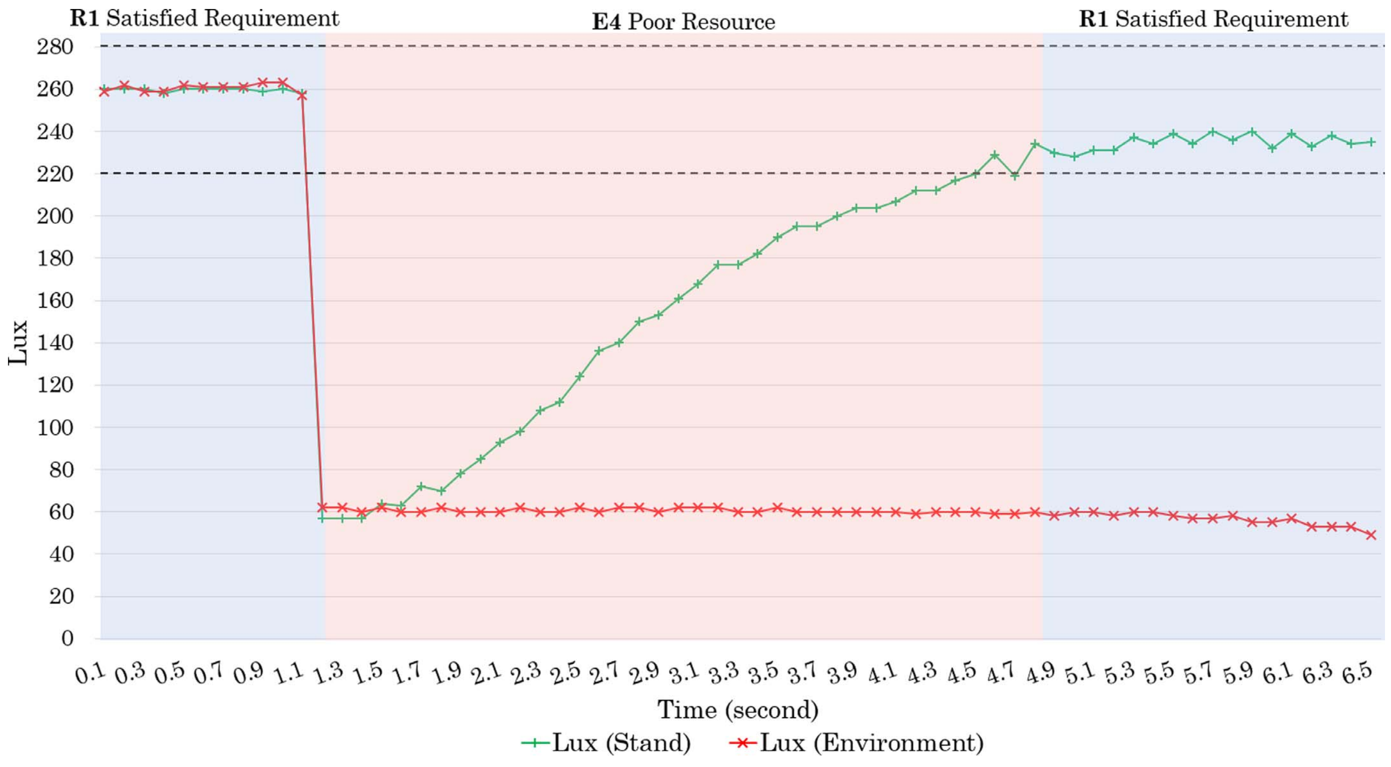


Fig. 24. Results of scenario #2.

to 4.2 s, even though the illumination sensor is broken.

In scenario #5, the application needs to adapt to a sudden increase in external light. As described above, Fig. 26 shows that the illumination sensor is broken at 2.3 s, and thus, the application uses the secondary sensor built in the phone. The system satisfies the requirement from 4.3 to 9.2 s, but the external light suddenly increases at 9.3 s; therefore, the application determines that the system state is E3 (natural light). Subsequently, the application checks t2 (light can be reduced) and triggers A2 (reduce light) continually until 11.8 s.

Fig. 27 shows the results of scenario #6. This scenario is needed for adaptation to a sudden decrease in the external illumination with a broken illumination sensor. As shown in Fig. 27, the environment illumination suddenly decreases from 139 to 58 lx at 5.6 s. Furthermore, the lamp's illumination sensor does not work at 1.3 s. In this instance, the application determines that the system state is E1 (disabled sensor) from the results of A-FSM, and checks t1 (available sensor exists). Then,

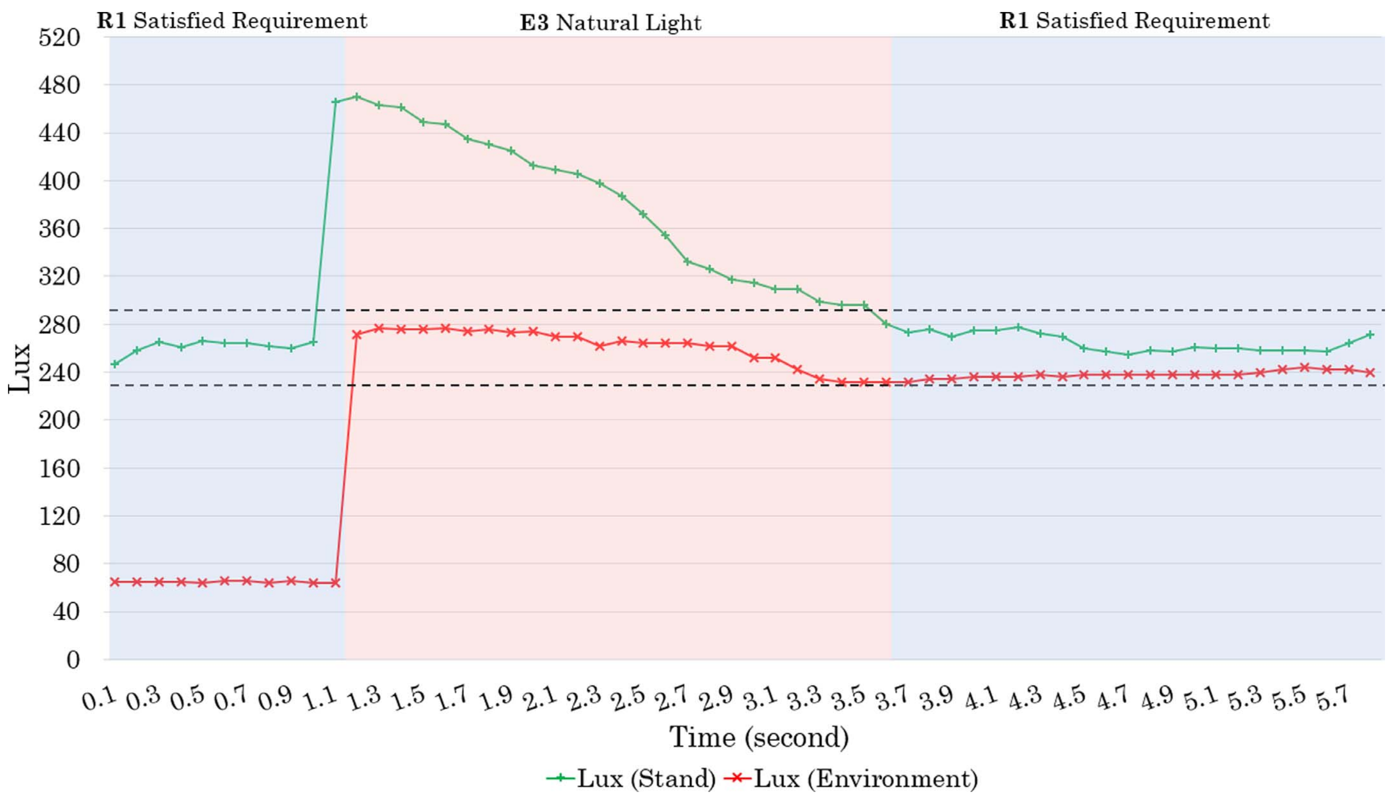


Fig. 25. Results of scenario #3.

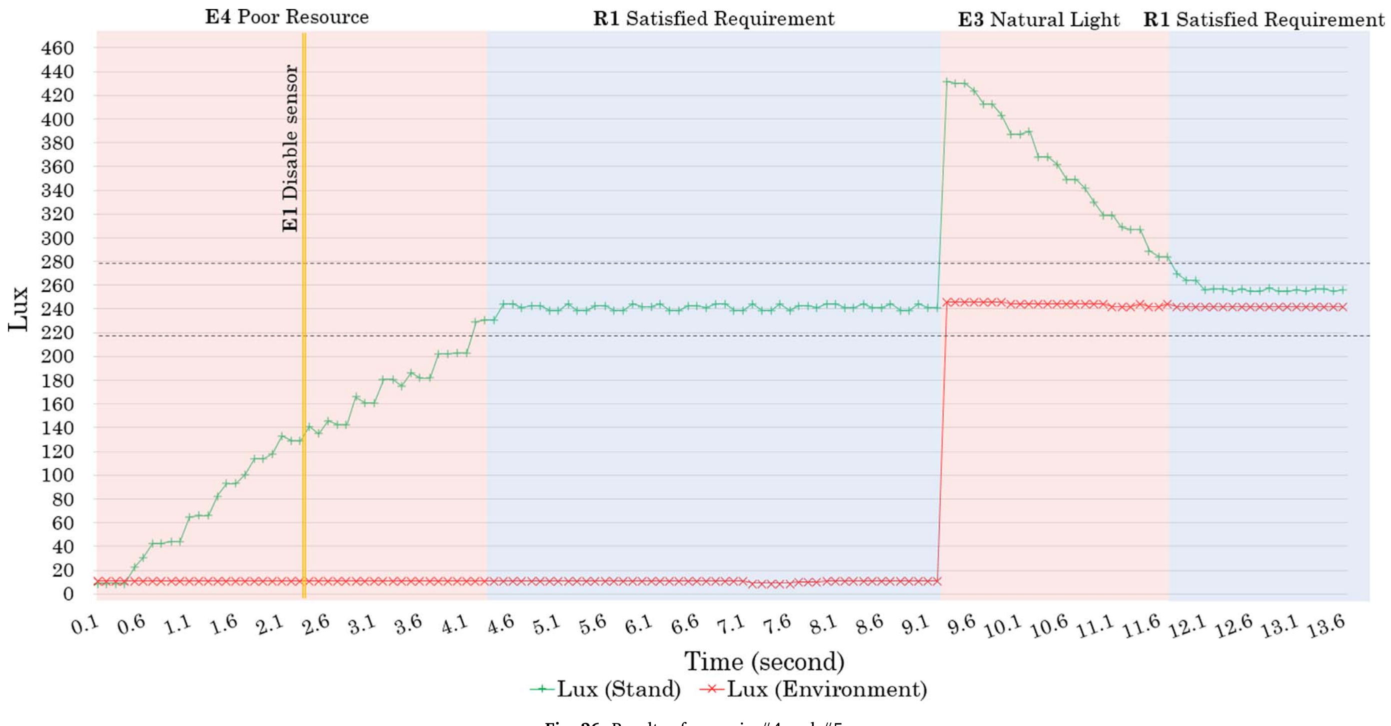


Fig. 26. Results of scenario #4 and #5.

the application adapts using the illumination sensors built into the phone. After adapting for E1 (disabled sensor), the application continues the adaptation process to resolve E4 (poor resource). The application checks t3 (light can be increased) and triggers A3 (increase light) until 7.1 s. Finally, the system adapts to the environment changes after 7.2 s.

The results of these scenarios show that the proposed framework performs reasonably well with regard to adaptiveness to various environment changes. Overall, the results of the scenarios show the practical usability of the proposed self-adaptive framework at runtime.

6. Discussion

In this section, we describe assumptions for software conditions of

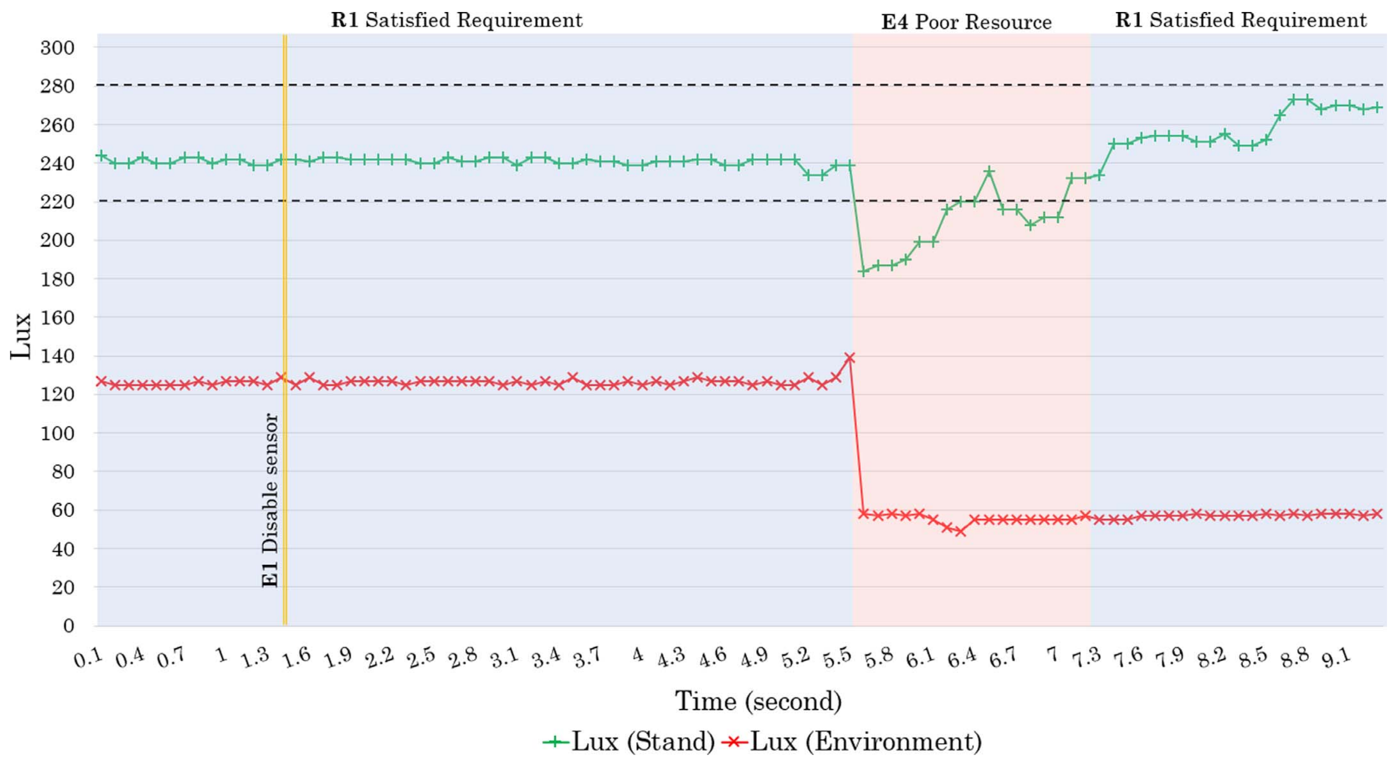


Fig. 27. Results of scenario #6.

RINGA. Furthermore, the limitations of RINGA and future work to overcome these limitations are described.

6.1. Assumptions for software conditions of RINGA

In this section, we describe assumptions for the software conditions that can be applied to RINGA, given the experimental results and limitations of the proposed approach. RINGA can be applied to software when

- the software is described as a finite state machine
- the software environment can be predictable, but its changes are not predictable at runtime
- fast analysis, planning, and execution are needed to adapt to environmental changes
- low computing power is available for verifying software at runtime

In this paper, we proposed a finite state machine (SA-FSM) to describe self-adaptive software. Thus, to apply the proposed approach, the software needs to be modeled as finite state machine. Further, the environment in which the software is executed should be predictable at design time, but its changes should not be predictable at runtime because the design of SA-FSM needs to contain strategies to adapt to environment changes. RINGA can adapt to environment changes even in devices with low computing power and has fast verification at runtime (Section 4.2). Moreover, we performed an experiment with a finite state machine consisting of less than 45 states and the extraction time of A-FSM grew exponentially (Section 4.1). Hence, the state number of designed finite state machine is restricted under 45 states for software stability. However, we assume that RINGA can be applied to more complex finite state models, if the predesign process is extended to optimize A-FSM. Moreover, in this paper, we only implemented a prototype of RINGA. However, we have a plan to implement a stable version to support the self-adaptive software development process.

6.2. Validity threats

In this section, we discuss the validity of the proposed framework for generalization. First, we performed the experiment with several cases of randomly generated well-formed finite state machines. These experimental finite state machines are used to evaluate design-time and runtime performance. We consider the experimental data to evaluate the worst case for the proposed method. In the experimental data set, every transition must be explored to abstract A-FSM from SA-FSM; therefore, abstraction results (i.e., $\rightarrow_{\text{A-FSM}}$) are also complex equation forms. The experimental results show that the proposed method is efficient at runtime. However, the experiment was conducted with limited states and transitions because it is not feasible to execute large-scale finite state machines in a mobile computing environment. Naturally, a high computing environment (e.g., workstation and server) can handle large-scale finite state machines than a mobile computing environment. Nevertheless, the high computing environment also has limitations in processing very large-scale finite state machines. Therefore, it is needed to process large-scale finite state machines for applying large-scale software. We discuss this problem in detail in the next section.

The other validity threat is related with the exemplar of the paper. We described an example application to show that RINGA can be applied to self-adaptive software at runtime. The exemplar uses a simple IoT scenario, and there are two devices (smartphone and light control). The exemplar simulated six scenarios, and the scenario results showed that RINGA can be adapted to various environment changes. However, the IoT environment is connected to various devices; therefore, the environment is more complex than the exemplar. In addition, other self-adaptive software environments will need more complex finite state machines if RINGA is applied. Therefore, the exemplar validates that simple software can be applied to the proposed framework, but large-scale software should be carefully approached. To address large-scale self-adaptive software, the proposed framework should improve the design of finite state machines. We discuss this limitation and future work in the next section. Finally, the experiment and the exemplar validate that the proposed framework can be applied to self-adaptive software at runtime; however, the framework needs improvement to be applied to large-scale software.

6.3. Limitations and future work

In this paper, we proposed the design of self-adaptive software based on finite state machines, and verified the design of self-adaptive software using a model-checking methodology at runtime. Moreover, we proposed finite state machines (SA-FSM and A-FSM) for this design. Even though the proposed approach yielded excellent experimental results, there are several problems to be solved before moving forward.

The first limitation is the complexity of a designed model. It is difficult to apply the proposed model to large-scale self-adaptive software. If a software design requires many states and transitions, it would be difficult to design using the proposed finite state machine models. To solve the complexity of designed model, we plan to improve finite state machine design. One idea is the division and integration of the finite state model. The idea is that the overall design of the finite state machine is divided into small parts, and each part is described as an SA-FSM. The partially designed finite state machines are then integrated into one SA-FSM. In addition, if this idea is possible, we assume that it would be possible to add, change, and delete finite state models in a predesigned model. Therefore, this idea can extend RINGA as incrementally expandable framework.

The second limitation is the increased computing power needed for the abstraction algorithm. To verify the self-adaptive software at runtime, the designed finite state model is abstracted in the form of equations that contain the extracted paths to reach adaptive ($S_{adaptive}$) and dissatisfied (S_{dis}) states in the design-time phase (Section 3.2.3). However, as the number of states or transitions increase, the abstraction (precomputing) and runtime process time grow exponentially (Sections 4.1 and 4.2). One reason for this is that the abstraction algorithm explores all possible ways to reach the adaptive and dissatisfied states. The other reason is that the length of an A-FSM transition also increases when number of states or transitions increase, so more computing is needed to calculate the larger A-FSM. These limitations also make it difficult to apply the proposed approach in large-scale software. Further, if a designed finite state machine is changed, the abstraction process is repeated, and this can consume computing resources. As mentioned above, we plan to improve the model of RINGA using the division and integration of finite state machines. In this idea, partitioned models partially perform the abstraction process. After the abstraction process of the partitioned models, each abstracted result can be merged as one abstracted result. We assume that it could reduce the abstraction process time by reducing overlapping calculations.

The other limitation is the limited expressiveness of the current RINGA requirements. RINGA involves requirements pertaining to the design of the SA-FSM, and the description of specific requirements is limited, for example, to time expressions, bounded iteration, and so on. Therefore, RINGA needs to express requirements related to time. To address the expression of time-related requirements, we plan to use linear temporal logic (LTL) to express requirements. LTL is a temporal logic model for modalities referring to time [21]. Rigorous rules are needed to apply LTL in RINGA. Moreover, the abstraction algorithm needs to be extended because RINGA currently only considers the reachability of the finite state machine. Therefore, it is necessary to develop an abstraction algorithm to express LTL with respect to time (e.g., “next” and “until”). Moreover, it could be possible to apply previous research to express requirements. For example, Nicolás et al. [36,37] proposed an approach to synthesize live behavior model. They applied labeled transition systems as models to describe the event based model, and fluent LTL to describe specifications. Their research produces a liveness model by considering the liveness assumption for the behavior of the environment and models a liveness goal for a system and the satisfaction of the system goal.

7. Conclusion

In this paper, we proposed a self-adaptive software framework with a finite state machine design and verification at runtime. The proposed framework has two parts: design-time and runtime. The design-time is responsible for designing the self-adaptive software with a finite state machine and precomputing it for runtime verification. To design self-adaptive software, we proposed SA-FSM, which is abstracted for runtime verification using an abstraction algorithm as A-FSM. A-FSM is used to verify the self-adaptive software at runtime. The runtime process is responsible for verification using the precomputed finite state machine (A-FSM) in a MAPE loop. We performed an empirical evaluation and implementation using an IoT-based example application. The empirical evaluation compared the proposed framework with two symbolic model checkers, and the results showed that, although the proposed method has a precomputation process, it saves time at runtime by considering several model-checking scenarios. Furthermore, we implemented Android and Arduino applications to measure its adaptability in a real environment. We proposed a simple IoT-based example application for adapting to illumination changes. We tested our method with six scenarios, and the results showed that the proposed framework can adapt to real environments. Moreover, we discussed the limitations of the proposed approach, and described future work for solving the discussed limitations.

For the future, we plan to extend our method to improve our finite state model and abstraction process. To improve model design, we will apply LTL. We assume that it will produce rich requirement expressions at design time. Moreover, to improve the abstraction algorithm, we plan to develop the designed finite state model in parts. We assume that if a finite state model can be abstracted and merged, the abstraction performance will be improved. Moreover, the partial model has the potential to expand the self-adaptive software incrementally. Finally, we will demonstrate the proposed approach in a complex IoT environment that consists of multiple devices.

Acknowledgements

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2012M3C4A7033346). This work also partially was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. 2016-0-00498, User behavior pattern analysis based authentication and anomaly detection within the system using deep learning techniques)

ANNEX. Acronyms

Acronyms	Full definition
A-FSM	Abstracted Finite State Machine
INVEST	Incremental VErification Strategy
IoT	Internet of Things
ISM	Interactive State Machine

MAPE	Monitoring-Analyzing-Planning-Executing
NuSMV	New Symbolic Model Checker
RINGA	Runtime verIfication with fiNite state machine desiGn for self-Adaptation software
SA-FSM	Self-Adaptive Finite State Machine
SMV	Symbolic Model Checker
SOTA	State of the Affairs

References

[1] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Autonom. Adapt. Syst. 4 (2) (2009) 14.

[2] R. De Lemos, et al., Software engineering for self-adaptive systems: a second research roadmap, Software Engineering for Self-Adaptive Systems II, Springer Berlin Heidelberg, 2013, pp. 1–32.

[3] R. Calinescu, C. Ghezzi, M. Kwiatkowska, R. Mirandola, Self-adaptive software needs quantitative verification at runtime, Commun. ACM 55 (9) (2012) 69–77.

[4] C. Baier, J.-P. Katoen, et al., Principles of Model Checking, MIT Press Cambridge, 2008.

[5] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Logic Algebr. Programm. (2009) 293–303.

[6] L. Tesei, E. Merelli, N. Paoletti, Multiple levels in self-adaptive complex systems: a state-based approach, Proceedings of the European Conference on Complex Systems 2012, Springer International Publishing, 2013, pp. 1033–1050.

[7] D.B. Abeywickrama, F. Zambonelli, Model checking goal-oriented requirements for self-adaptive systems, Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on, IEEE, 2012, pp. 33–42.

[8] J. Cámara, R. De Lemos, Evaluation of resilience in self-adaptive systems using probabilistic model checking, Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE Press, 2012, pp. 53–62.

[9] K. Johnson, R. Calinescu, S. Kikuchi, An incremental verification framework for component-based software systems, Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering, ACM, 2013, pp. 33–42.

[10] A. Filieri, C. Ghezzi, G. Tamburrelli, Run-time efficient probabilistic model checking, Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 341–350.

[11] A. Filieri, G. Tamburrelli, Probabilistic verification at runtime for self-adaptive systems, Assurances for Self-Adaptive Systems, Springer Berlin Heidelberg, 2013, pp. 30–59.

[12] A. Filieri, G. Tamburrelli, C. Ghezzi, Supporting self-adaptation via quantitative verification and sensitivity analysis at run time, IEEE Trans. Softw. Eng. 42 (1) (2016) 75–99.

[13] W Yang, C Xu, Y Liu, C Cao, X Ma, J Lu, Verifying self-adaptive applications suffering uncertainty, Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM, 2014, pp. 199–210.

[14] H.A. Müller, H.M. Kienle, U. Stege, Autonomic computing now you see it, now you don't, In Software Engineering, Springer, 2009, pp. 32–54.

[15] A. Computing et al. An architectural blueprint for autonomic computing. IBM White Paper, 2006.

[16] J.O. Kephart, D.M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[17] G. Tallabaci, V.E. Silva Souza, Engineering adaptation with Zanshin: an experience report, Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE Press, 2013, pp. 93–102.

[18] C. Barna, et al., Hogna: a platform for self-adaptive applications in cloud environments, 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE, 2015, pp. 83–87.

[19] D. Garlan, et al., Rainbow: architecture-based self-adaptation with reusable infrastructure, Computer 37 (10) (2004) 46–54.

[20] E.M. Fredericks, B. DeVries, B.H.C. Cheng, Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty, Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, ACM, 2014, pp. 17–26.

[21] A. Knauss, et al., ACon: a learning-based approach to deal with uncertainty in contextual requirements at runtime, Inf. Softw. Technol. 70 (2016) 85–99.

[22] Y. Wang, J. Mylopoulos, Self-repair through reconfiguration: a requirements engineering approach, Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2009, pp. 257–268.

[23] J. Wuttke, et al., Traffic routing for evaluating self-adaptation, Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE Press, 2012, pp. 27–32.

[24] D. Weyns, R. Calinescu, Tele assistance: a self-adaptive service-based system exemplar, Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE Press, 2015, pp. 88–92.

[25] J.E. Hopcroft, Introduction to Automata Theory, Languages, and Computation, Pearson Education India, 2007.

[26] Samsung, Electronics & appliances: tablets, Smartphones, TVs | Samsung US http://www.samsung.com/, 2016 (Accessed: 2016. 9. 9).

[27] Intel | Data Center Solutions, IOT, and PC innovation, 2017 (Accessed: 2017. 5. 11). http://www.intel.com/.

[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an open source tool for symbolic model checking, Computer Aided Verification, Springer, 2002, pp. 359–364.

[29] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, NuSMVv 2.2 tutorial, ITC-irst-Via Sommarive 18 (2004) 38055.

[30] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A new symbolic model verifier, Computer Aided Verification, Springer, 1999, pp. 495–499.

[31] NuSMV, NuSMV: a new symbolic model checker, http://nusmv.fbk.eu/, 2016 (Accessed: 2016. 9. 9).

[32] The cadence SMV model checker, http://www.kenmcmil.com/smv.html, 2016 (Accessed: 2016. 9. 9).

[33] Fritzing, 2016 (Accessed: 2016. 9. 9). http://fritzing.org/.

[34] Arduino, Arduino – ArduinoBoardUno, https://www.arduino.cc/en/Main/ArduinoBoardUno, 2016 (Accessed: 2016. 9. 9).

[35] DFROBOT, Bluno –Arduino UNO with bluetooth 4.0(BLE) DFRobot, http://www.dfrobot.com/index.php?route=product/product&product_id=1044#.V2eUQfmLSrw, 2016 (Accessed: 2016. 9. 9).

[36] D. Nicolás Roque, et al., Synthesis of live behaviour models, Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2010, pp. 77–86.

[37] D. Nicolás Roque, et al., Synthesis of live behaviour models for fallible domains, Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 211–220.