Self-Adaptive Framework with Game Theoretic Decision Making for Internet of Things

Euijong Lee
Department of Computer and Information Security
Sejong University
Seoul, Republic of Korea
kongjjagae@sejong.ac.kr

Young-Gab Kim*
Department of Computer and Information Security
Sejong University
Seoul, Republic of Korea
alwaysgabi@sejong.ac.kr

Young-Duk Seo
Department of Computer and Information Security
Sejong University
Seoul, Republic of Korea
mysid88@sejong.ac.kr

Doo-Kwon Baik
Department of Computer Science and Engineering
Korea University
Seoul, Republic of Korea
baikdk@korea.ac.kr

***Abstract*—The Internet of Things (IoT) connects several objects within environments that dynamically change, and so requirements may be added and changed at runtime. Therefore, requirements may be satisfied at dynamic change. Self-adaptive software can alter their behavior to satisfy requirements in dynamic environments. In this perspective, the concept of self-adaptive software is suitable for IoT environments. In this study, a self-adaptive framework is proposed for decision making in IoT environments at runtime. The framework includes finite-state machine model designs and game theoretic decision-making methods to extract efficient strategies. The framework is implemented as a prototype, and experiments are performed to evaluate runtime performance. The results demonstrate that the proposed framework can be applied to IoT environments at runtime.**

***Index Terms*—Self-adaptive software, Game theory, Finite-state machine, Nash equilibrium, Internet of Things**

I. Introduction

Internet of Things (IoT) technologies are currently widespread. IoT connects several objects within various dynamic environments [1]. The IoT environment includes several requirements, and the requirements must be dynamically satisfied at runtime. Therefore, IoT frameworks should be able to make decisions that satisfy the requirements of IoT environments [2]. From this perspective, self-adaptive software may be applied within IoT frameworks. Self-adaptive software can change their behavior or structure within changing environments at run-time [3], [4]. For this reason, self-adaptive software is appropriated for dynamic IoT environments. However, previous studies have highlighted several limitations of self-adaptive software. These limitations include the requirement that models of self-adaptive software should contain expected adaptive strategies to adapt to changes in environmental conditions. Therefore, such models are not suitable for IoT environments with changing components at runtime. In addition, IoT has several requirements between different objectives, and an effective decision-making method is required. In this perspective, game theory may be used for decision making between different requirements. Game theory is a mathematical model for decision making between different stakeholders [5], [6]. It is applied in economics, biology, and computer science [6]–[11]. Game theory basically helps find optimized decisions. Therefore, game theoretical methods can be used in IoT environments to find the optimal decision between different requirements. In this paper, we propose a self-adaptive framework for IoT with game theoretical strategy extraction methods and finite-state machine designs. The design of a finite-state machine is based on prior studies [12], [13].

The remainder of this paper is organized as follows. Section 2 provides a background of self-adaptive software and game theory, and related work that describe self-adaptive software as a finite-state machine are also presented. In Section 3, the proposed framework is introduced. In Section 4, the empirical evaluation is presented. Section 5 concludes the paper.

II. Background and Related Work

A. Self-Adaptive Software Framework

Self-adaptive software detects environmental conditions and changes its behavior or structure if software requirements are violated [3]. Therefore, a self-adaptive software includes a monitoring process to observe environmental changes, including its own condition. In addition, self-adaptive software analysis is related to adaptation using monitoring data. If adaptation is needed, adaptation strategies are developed and executed. This adaptation process is referred to as MAPE-loop and is used in self-adaptive software and autonomic computing [3], [4], [12]–[19].

The loop consists of four parts as follow:

- A monitoring process is responsible for collecting and correlating data from the environment and software.

* Corresponding author

- An analysis (detection) process is responsible for analyzing adaptive symptoms by monitoring data.
- A planning (deciding) process is responsible for determining what is needed to be changed and how to change.
- An execution (acting) process is responsible for applying an adaptation strategy.

The MAPE-loop is generally used in self-adaptive software and autonomic computing [3]. Therefore, several self-adaptive software studies have applied the loop [3], [4], [12]–[18], [20], and the proposed framework also uses the MAPE-loop.

B. Finite-State Machine for Self-Adaptive Software Modelling

In this study, a finite-state machine model is used to describe and verify self-adaptive software, and the finite-state machine is based on previous studies [12], [13]. Lee et al proposed a framework with finite-state machine to describe self-adaptive software (i.e., SA-FSM). The finite-state machine is translated as an abstracted model in equation form for runtime verification (i.e., A-FSM). The translation process contains abstraction algorithms that are based on state elimination. The abstracted model is used for runtime verification within a MAPE-loop. The framework produces reasonable experimental results and provide guidelines for modeling self-adaptive software by a finite-state machine. This study is modified at present study.

C. Nash Equilibrium

Nash Equilibrium was introduced by John Forbes Nash Jr [21], and it is used to analyze the results of strategic interactions among diverse decision makers. Furthermore, every finite game has a Nash equilibrium. Therefore, if there are several decision makers and institutions, the Nash equilibrium can be used to make forecasts [6]. In game theory, there are non-cooperative players who participate in a game, and they have their own strategies for different actions. The selection of strategies can affect the strategy of other players. Each player strives to achieve an outcome with the largest possible payoff. Therefore, players choose their strategy so that an outcome with maximum payoff may be obtained.

If players are in Nash equilibrium, then any player can select a better unilateral strategy. In Nash equilibrium, no one can receive a better payoff by changing strategies, and each strategy leads to the best outcome. Therefore, no players change their strategies, and strategies are solidified [5], [21]. In the proposed approach, Nash equilibrium is used to extract a strategy for adapting to an IoT environment. Details on the use of Nash equilibrium are discussed in Section 3.C.

III. Proposed Approach

A self-adaptive software framework is proposed for designing an IoT environment using finite-state machine and by extracting an adaptive strategy using Nash equilibrium. Section 3.A presents an overview of the proposed method. Section 3.B presents the modelling of finite-state machine based on SA-FSM [12], [13]. Section 3.C explains the method for extracting a strategy using Nash equilibrium.

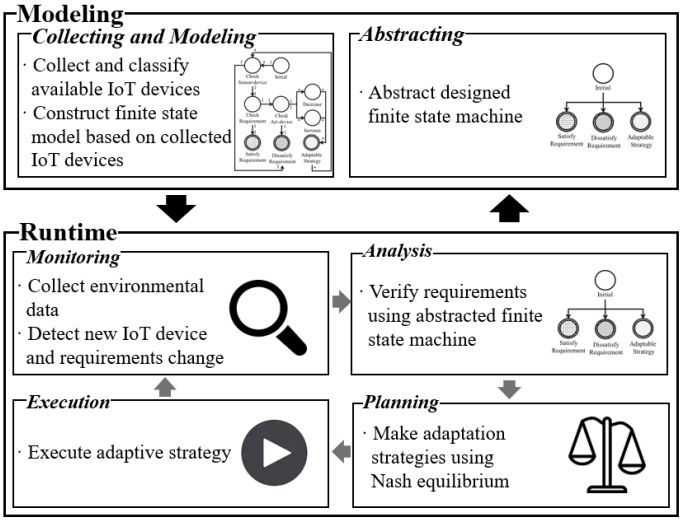

Fig. 1. Overview of proposed framework

A. Overview

In this study, a self-adaptive software framework is proposed for an IoT environment, and the framework includes two phases: modeling and runtime. The modeling phase is responsible for extracting finite-state machine to describe the self-adaptive software. The runtime phase includes the MAPE-loop and is responsible for adaptation at runtime. Fig. 1 presents an overview of the proposed framework.

As mentioned earlier, the modeling phase is responsible for building a model of an IoT environment as finite-state machine. The modeling phase first collects available IoT devices and prepares the model building process. Collected IoT devices are classified into two types: sensor device or act device. The classified devices are categorized based on their ability and related requirements. Details on the classification of devices are presented in Section 3.B.1. After classification, a finite-state machine is constructed using collected IoT devices. The finite-state machine model is constructed by action and relations of collected devices. Details on modeling a finite-state machine for IoT is described in Section 3.B.2. The final process of the modeling phase is the abstracting process. The abstracting process abstracts a designed finite-state machine using a state elimination algorithm [22]. Details on the abstraction algorithms are described in previous studies [12], [13]. Note that the abstracting process is only operated once if there is no change in the design of the finite-state machine. Finally, the abstracted finite-state machine is transferred to the runtime phase. The designed and abstracted models are used for evaluating the environmental condition of the software in each cycle of the MAPE-loop.

The runtime phase is responsible for adaptation at runtime. As we mentioned earlier, the runtime phase includes the MAPE-loop. The monitoring process is responsible for collecting data that describes the environment and internal changes. The environmental data is collected through sensor-

devices. In addition, the monitoring process searches for any new device that has been potentially added in the finite-state machine. Remodeling is requested if a new device is detected. If the modelling request is accepted, the modelling phase is executed and remodeling is performed with the new device. After the monitoring process, the analysis process is executed. The analysis process is responsible for analyzing symptoms that are related to the adaptation situation. In the proposed approach, analysis is performed only through calculating equations (i.e., A-FSM [12], [13]). Using the calculating equations, the analysis process determines whether the self-adaptive software has satisfied requirements and detects the conditions for adaptation. Analysis results are transferred to the planning process. The planning process is responsible for making adaptation strategies that must be changed and for determining how to affect those changes. In this study, decision making using Nash equilibrium is proposed, and the method can make strategies for adapting to environmental changes. Details on strategy extraction are described in Section 3.C. The last process of the runtime phase involves the execution that is responsible for activating the adaptive strategies. Therefore, if the planning process transferred the adaptive strategy, the execution process activates the adaptive strategy for adaptation. Subsequently, the monitoring process is executed, and the MAPE-loop continues.

B. Finite-State Machine Modeling for Self-Adaptive Software in an IoT Environment

We classified IoT devices as sensor-devices and act-devices for modelling IoT-based self-adaptive software. In this section, the classification of IoT-devices is presented, and the definition of finite-state machine modelling using the classification of IoT devices is described.

1) IoT Device Classification: There can be diverse devices in an IoT environment (light sensor, humidity sensor, light controller, speaker, humidifier, etc.). The devices can be classified as various categories. However, only two types of IoT devices are used for finite-state machine modelling. One is the sensor-device and the other is the act-device

- *Sensor-device* is used to sense environmental changes. Therefore, it should be embedded in at least one readable sensor device (light sensor, humidity sensor, temperature sensor, etc.) In addition, it is assumed that sensor-devices recognize which requirement is related to sensed data.
- *Act-device* is used to change the environment. Therefore, an act-device should be embedded in at least one physical device (LED, servo motor, fan, etc.) In addition, it is assumed that the act-device recognizes which requirements are related to its operation.

2) Design of Finite-State Machine for IoT: In this study, a finite-state machine is used to model self-adaptive software for IoT, and the finite-state machine is based on SA-FSM [12], [13]. However, SA-FSM is modified for IoT, and it is a tuple (S, , s0, AP, L), where

- S is a set of states.
- States are classified into eight types $\{S_{sensor}, S_{req}, S_{sat}, S_{dis}, S_{act}, S_{adapt}, S_{inc}, S_{dec}\} \subseteq S$.
- $S_{dis}, S_{sat}, S_{adapt}$ are end states.
- $\rightarrow \subseteq S \times S$ is the transition relation and it is classified as eleven types $\{s_0 \times S_{sensor}, S_{sensor} \times S_{dis}, S_{sensor} \times S_{req}, S_{req} \times S_{sat}, S_{req} \times S_{act}, S_{act} \times S_{dis}, S_{act} \times S_{inc}, S_{act} \times S_{dec}, S_{inc} \times S_{adapt}, S_{dec} \times S_{adapt}, S_{adapt} \times S_{sensor}\}$.
- s_0 is an initial state.
- AP is a set of atomic propositions.
- L: S $\rightarrow$ 2^{AP} is a labeling function (2^{AP} denotes the power set of AP).

As represented by the tuple definition, the proposed finite state machine consists of nine states and eleven transitions types. The state set and related transitions are given as follows.

- *Initial state* (s_0) is an initial state.
- *Sensor-device state* (S_{sensor})is the set of sensor-device related states. In this state, sensor-devices must be related at least once. If a readable sensor device is available, it satisfies a requirement state (i.e., $S_{sensor} \times S_{req}$), but it is connected to a dissatisfied state (i.e., $S_{sensor} \times S_{dis}$) if there is no related sensor-device.
- *Requirement state* (S_{req})) is the set of states that verify requirement satisfaction. A satisfied state is reached if the checked requirement is satisfied (i.e., $S_{req} \times S_{sat}$), or an adaptive state is reached if the requirement is not satisfied (i.e., $S_{req} \times S_{adapt}$).
- *Satisfied state* (S_{sat}) is the set of end states in which the software requirements are satisfied.
- *Dissatisfied state* (S_{dis})) is the set of end states in which the software requirements are not satisfied. If the finite-state machine has no readable device (i.e., $S_{sensor} \times S_{dis}$) or no possible adaptive action (i.e., $S_{act} \times S_{dis}$), the finite-state machine model reaches this state.
- *Act state* (S_{act}) is the set of states that check for actable devices. If there are no actable devices, a dissatisfied state (i.e., $S_{act} \times S_{dis}$) is reached, or if there are actable devices, an increase or decrease state (i.e., $S_{act} \times S_{dis}$ and $S_{act} \times S_{inc}$) is reached.
- *Increase state* (S_{inc}) and *decrease state* (S_{dec}) are the set of act-device related states. In these states, at least one actable device is related. If the finite-state machine reaches such states, related act-devices are operated. The state then reaches an adaptive state (i.e., $S_{inc} \times S_{adapt}$, $S_{dec} \times S_{adapt}$).
- *Adapt state* (S_{adapt}) is one of the end state sets, which denotes possible adaptive activities. Therefore, if the finite-state machine reaches this state, it implies that the self-adaptive software must adapt, and there are several adaptive strategies. In addition, this state satisfies related requirements for the sensor-device state to recheck requirement satisfaction (i.e., $S_{adapt} \times S_{sensor}$).

In the proposed approach, reachable paths are extracted to reach end states (S_{sat}, S_{dis}, and S_{adapt}). In addition, reachable paths are calculated at each MAPE-loop cycle.

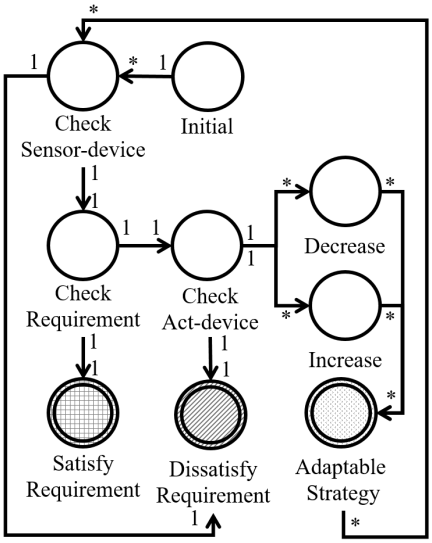

Fig. 2. Relationship of IoT-FSM state types

In addition, reachable paths are calculated at each MAPE-loop to verify requirement satisfaction. Fig. 2 presents a graphical definition of the proposed finite-state machine. As shown in Fig. 2, most of the transition is matched one-on-one. However, there are some transitions that do not experience one-on-one matching, and the reason is described below. Initial state can relate to multiple S_{sensor} states, which implies that an initial state can be reaches several requirement states. In addition, an S_{act} state can be related to multiple S_{inc} and S_{dec} states because it is possible to have multiple act-devices in a single requirement. For example, if a requirement is illumination, there are several possible act-devices that can exist (LED, natural light etc.). Furthermore, S_{inc} and S_{dec} states can be connected to multiple Sadapt because the operation of an act-device can affect several requirements. For example, if an act-device can control windows, it is possible to affect multiple requirements (light, temperature, intensity of dust, etc.).

C. Game Theoretic Decision Making

This section describes how Nash equilibrium is used for strategy extraction from finite-state machine modelling. As described before, an act-device can perform physical acts, and the act-device can affect several requirements. In other words, a requirement can have several act-devices and the related act-devices can affect other requirements. Therefore, act-devices can be operated to satisfy requirements, and the operation of an act-device can affect several requirements. In addition, if requirements are related to overlapped act-devices, it is possible that an act-device adversely affects different requirements. In this case, one requirement is satisfied, but the other is not. Therefore, the operation of strategies must ensure that act-devices are operated effectively to satisfy multiple requirements. In this regard, requirements can be considered a player, and an act-device can be considered the strategy of the player. The Nash equilibrium for IoT is described below:

Let a player be a requirement, and let (S, f) be a game with n requirements, where

- $S = S_1 \times S_2 \times \cdots \times S_n$ is a strategy set of profile.
- Requirement $i \in \{1, \cdots, n\}$.
- $f(x) = \{f_1(x), \cdots, f_n(x)\}$ is the payoff function.
- A payoff function is evaluated at $x \in S$.
- x_i is an act-device profile of requirement i.
- x_{-i} is an act-device profile of the other requirements.
- Requirement i operates act-device x_i resulting in strategy profile $x = (x_1 \cdots x_n)$; then, requirement i obtains payoff $f_i(x)$.
- $x^* \in S$ is a Nash equilibrium for IoT when $\forall i, x_i \in S_i$: $f_i(x_i^*, x_{-i}^*) \geq f_i(x_i, x_{-i}^*)$.
- x^* is the operation candidate at runtime, and
- A strategy with the highest Nash equilibrium value among requirements is selected and implemented.

Formally, if players (requirements) reach Nash equilibrium, they cannot choose a new strategy because no one can receive better payoffs by selecting a new strategy. However, Nash equilibrium in the proposed model indicates candidate operations because the Nash equilibrium denotes that there are strategies that can satisfy multiple requirements. However, the proposed model is required as a method to evaluate and select the most effective strategy, and the method is described in Section 3.D.

D. Strategy Evaluation

As mentioned earlier, the proposed Nash equilibrium model denotes candidate strategies in an IoT environment. However, it is possible that there are multiple candidate strategies. In this case, strategies should be evaluated for the selection of an optimal strategy. Therefore, a method that evaluates strategies (Nash equilibrium) is proposed, and it is called a strategy score (SS). There are three considerable conditions and they are described below:

- The *number of satisfied requirements* (SR) is the number of requirements that may be satisfied by strategy execution. If an adaptation strategy satisfies multiple requirements, it is more efficient than an adaptive strategy that satisfies less requirements.
- The *number of related requirements* (RR) is the number of requirements that may be affected by the execution of a strategy. For example, if a strategy opens the windows for adjusting indoor brightness, it affects humidity, dust density, and temperature. In this case, requirements for humidity, dust density, and temperature are RR. Having fewer RR is efficient.
- The *number of act devices* (AD) is the number of act-devices that are executed by an adaptation strategy. Smaller is more efficient.

Equation 1 presents the calculation of SS using SR, RR, and AD.

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

The equation includes the sum of two terms: requirement and act-device. The first term (i.e., $\left\{\log\left(\frac{SR+1}{RR+1}+1\right)\right\}$) is the related requirement, and so SR and RR are used for the term. As we mentioned earlier, it is efficient when a strategy satisfies several requirements (i.e., large value of SR), and when a strategy affects few requirements (i.e., small value of RR). Therefore, SR is divided by RR, and assumes a logarithmic function for normalization. 1 is added to prevent a negative infinity output. The second term (i.e., $\left\{\log\left(\frac{1}{AD+1}+1\right)\right\}$) is related to act-devices. If a strategy can satisfy the same requirements, a lower AD value is more efficient. Therefore, a reciprocal number of AD is used, and 1 is added to prevent a negative infinity output. The terms α and β are mediators used for adjusting the power of each term. The equation denotes the strategy score of a strategy within the planning process of the MAPE-loop.

IV. Experiment

A prototype of the proposed framework using JAVA 1.8.0 was implemented on an Intel Core i5-4670 CPU (3.4 GHz) PC with 16 GB RAM memory. The purpose of the experiment was to evaluate the performance of the proposed framework in various environments. Random generation of IoT environments was conducted using different numbers of act-devices and requirements. A requirement had at least one sensor-device and act-device, and the remaining act-devices were randomly assigned to requirements. In addition, environment values (i.e., sensed data) were randomly varied and iterated 100 times for each experiment. Three factors were measured: abstracting time, analysis time, and planning time.

The first experiment involved ten fixed requirements and variable act-devices. Fig. 3 shows the experimental results. Fig. 3 shows the results obtained for increasing numbers of act-devices. Naturally, the method required more time for a larger number of act-devices. However, the maximum average time for abstracting IoT-FSM was less than 1.5 ms, and the analysis time less than 6 ms with 50 act-devices. Particularly, planning time for extracting strategies was higher than other method, but less than 300 ms even with 50 active devices. In addition, loop time (the sum of analysis and planning time) was less than 305 ms with 50 act-devices and ten requirements. Monitoring and executing time were ignored because the prototype neither read real sensor values nor operated real physical devices. Nevertheless, it was assumed that every factor was calculated within reasonable time.

The second experiment was conducted using 40 act-devices and variable requirements. Fig. 4 shows the experimental results. As in the previous experiment, the abstracting process required more time when the number of requirements was increased. However, the analysis and the planning processes time shows a tendency to decrease after rising. The reason

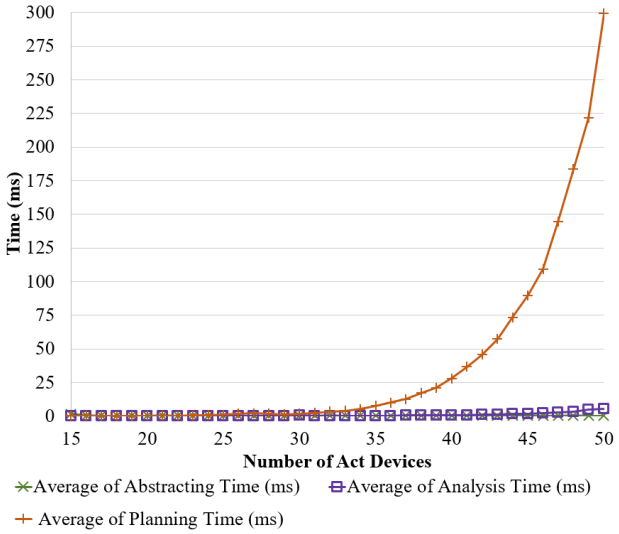

Fig. 3. Result with fixed requirements and increasing act- devices.

for this is that the interconnections of the requirements were more complicated. To analyze the IoT-FSM, the IoT-FSM is abstracted to an equation (See Sections 2.B and 3.A). The abstracted equation is complicated when the interconnection of requirements from the IoT-FSM is complicated. In addition, to extract Nash equilibrium, the possible actions for a requirement are compared with possible actions for other requirements. By contrast, extracting Nash equilibrium requires less time when the interconnections of the requirements are not complicated (i.e., the result of 2 and 30 requirements in Fig. 4). Therefore, the results of the second experiment show that analysis and the planning time are affected by the complexity rather than the number of requirements. Nevertheless, the second experiment also shows that the loop-time of the proposed approach is reasonable even when the requirement interconnections are complicated.

V. Conclusion

An IoT framework includes several requirements for accomplishing different objectives within changing environments, and the requirements should be dynamically satisfied at runtime. To solve this problem, a self-adaptive framework with strategy extraction in an IoT environment at runtime was proposed. The proposed framework consists of two phases: modelling and runtime. The modelling phase is responsible for searching available IoT devices and building a system model. To build a system model, a finite-state machine was proposed. After the modelling process, the abstracting process was performed to abstract the built model in equation form using state elimination algorithm. The abstracted results are transferred to a runtime phase. The runtime phase consists of an MAPE-loop. In the monitoring part, the environment date is obtained from available sensor-devices, and the data is transferred for the analysis process. The analysis process calculates equations that are extracted in the modelling phase and verifies requirements satisfaction at runtime. In the

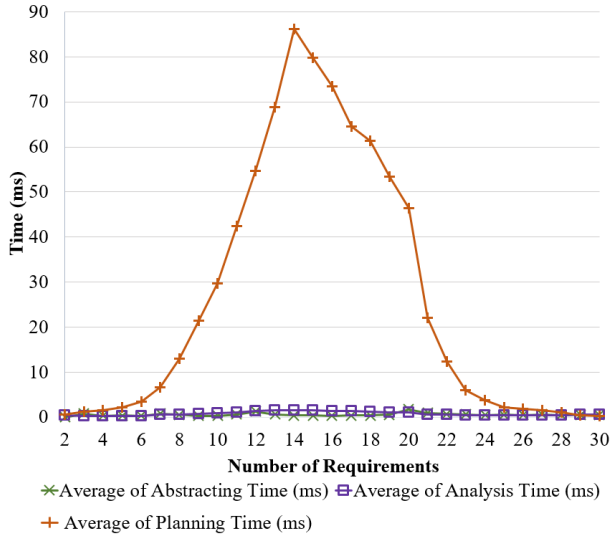

Fig. 4. Result with fixed act-devices and increasing requirements.

planning process, adaptive strategies are extracted using the proposed Nash equilibrium. The strategies are evaluated in the planning process, and the most efficient strategy is executed in the execution process. In this study, we demonstrate the suitability of the proposed framework. The results of the experiments demonstrate that the proposed extracting strategy can be applied in runtime. In this study, the suitability of the proposed framework was demonstrated by experiments that illustrated that the proposed extraction strategy can be applied at runtime, yielding reasonable results in terms of computation time.

In future work, optimization of the proposed method will be considered, particularly the planning process, for extension to mobile computing environments. In addition, the proposed framework should be applied in a real physical environment. Therefore, we will implement a physical IoT environment [23], [24] and appliy the proposed framework.

Acknowledgment

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2016-0-00498, User behavior pattern analysis based authentication and anomaly detection within the system using deep learning techniques) and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2017-0-00756, Development of interoperability and management technology of IoT system with heterogeneous ID mechanism).

References

[1] A. Rayes and S. Samer, "Internet of thingsfrom hype to reality," *The road to Digitization. River Publisher Series in Communications, Denmark*, vol. 49, 2017.

[2] S. Balasubramaniam and R. Jagannath, "A service oriented iot using cluster controlled decision making," in *Advance Computing Conference (IACC), 2015 IEEE International*. IEEE, 2015, pp. 558–563.

[3] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM transactions on autonomous and adaptive systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

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

[5] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic game theory*. Cambridge University Press Cambridge, 2007, vol. 1.

[6] Y. Shoham, "Computer science and game theory," *Communications of the ACM*, vol. 51, no. 8, pp. 74–79, 2008.

[7] M. Bhatia and S. K. Sood, "Game theoretic decision making in iot-assisted activity monitoring of defence personnel," *Multimedia Tools and Applications*, vol. 76, no. 21, pp. 21 911–21 935, 2017.

[8] X. Tao, G. Li, D. Sun, and H. Cai, "A game-theoretic model and analysis of data exchange protocols for internet of things in clouds," *Future Generation Computer Systems*, vol. 76, pp. 582–589, 2017.

[9] P. Semasinghe, S. Maghsudi, and E. Hossain, "Game theoretic mechanisms for resource management in massive wireless iot systems," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 121–127, 2017.

[10] F. Azzedin and M. Yahaya, "Modeling bittorrent choking algorithm using game theory," *Future Generation Computer Systems*, vol. 55, pp. 255–265, 2016.

[11] J. Zheng, Y. Cai, X. Chen, R. Li, and H. Zhang, "Optimal base station sleeping in green cellular networks: A distributed cooperative framework based on game theory," *IEEE Transactions on Wireless Communications*, vol. 14, no. 8, pp. 4391–4406, 2015.

[12] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D.-K. Baik, "Runtime verification method for self-adaptive software using reachability of transition system model," in *Proceedings of the Symposium on Applied Computing*. ACM, 2017, pp. 65–68.

[13] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D. Baik, "Ringa: Design and verification of finite state machine for self-adaptive software at runtime," *Information and Software Technology*, vol. 93, pp. 200–222, 2018.

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[15] A. Knauss, D. Damian, X. Franch, A. Rook, H. A. Müller, and A. Thomo, "Acon: A learning-based approach to deal with uncertainty in contextual requirements at runtime," *Information and software technology*, vol. 70, pp. 85–99, 2016.

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

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

[18] W. Yang, C. Xu, Y. Liu, C. Cao, X. Ma, and J. Lu, "Verifying self-adaptive applications suffering uncertainty," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 199–210.

[19] E. Lee and D.-K. Baik, "A verification technique for self-adaptive software by using model-checking," in *Proceedings on the International Conference on Artificial Intelligence (ICAI)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015, p. 395.

[20] Y.-D. Seo, Y.-G. Kim, E. Lee, K.-S. Seol, and D.-K. Baik, "Design of a smart greenhouse system based on mape-k and iso/iec-11179," in *Consumer Electronics (ICCE), 2018 IEEE International Conference on*. IEEE, 2018, pp. 1–2.

[21] P. D. Straffin, *Game theory and strategy*. MAA, 1993, vol. 36.

[22] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

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

[24] J. Lee, E. Lee, and D.-K. Baik, "Simulation and performance evaluation of the self-adaptive light control system," *Journal of the Korea Society for Simulation*, vol. 25, no. 2, pp. 63–74, 2016.