Runtime Verification Method for Self-Adaptive Software using Reachability of Transition System Model

Euijong Lee
Korea University
Seoul, Republic of Korea
kongjjagae@korea.ac.kr

Young-Gab Kim*
Sejong University
Seoul, Republic of Korea
alwaysgabi@sejong.ac.kr

Young-Duk Seo
Korea University
Seoul, Republic of Korea
seoyoungd@korea.ac.kr

Kwangsoo Seol
Korea University
Seoul, Republic of Korea
seolks@korea.ac.kr

Doo-Kwon Baik*
Korea University
Seoul, Republic of Korea
baikdk@korea.ac.kr

ABSTRACT

Self-adaptive software can change its own behavior in order to achieve an intended objective in a changing environment. Consequently, self-adaptive software requires practical runtime verification and validation. We propose an approach for runtime verification of self-adaptive software by using a designed transition system model. The proposed approach consists of two phases: pre-computing phase and runtime phase. In the pre-computing phase, we assume that the self-adaptive software is designed as a transition system. In this phase, the proposed approach translates the designed transition system into equations for runtime verification. For translation, we suggest an algorithm based on state elimination and reachability. After the pre-computing phase, the results of the translated equations are verified in the runtime phase. In order to demonstrate the suitability of our proposed approach, we performed experiments to evaluate the performance of the pre-processing phase and the runtime phase. In comparison with other model-checking tools, our approach achieved excellent results.

CCS Concepts

• Software and its engineering → Software verification and validation • Software and its engineering → Model-driven software engineering

Keywords

Self-adaptive software, transition system, model checking.

[*] Corresponding authors

1. INTRODUCTION

In recent times, the emergence of various software platforms has resulted in a varied software environment. Further, owing to developments in mobile devices and Internet of Things (IoT), software systems must operate in various environments. Self-adaptive software aims to change its own behavior or structure in a changing environment at runtime [1]. The characteristics of self-adaptive software make it suitable for adoption in the current scenario that requires software to operate in various environments. Design model and verification are important elements in self-adaptive software. Further, the demand for research related to practical verification at runtime has increased [2, 3]. Model checking is an effective static verification method to verify software described by a transition system [2-4]. Although model checking demonstrates excellent verification performance, it has a major problem known as the state explosion problem. However, in spite of this limitation, the method has been applied in various studies to verify self-adaptive software [5-10]. The challenges in earlier studies include unsuitability at runtime, state explosion problems, or the absence of a mechanism to apply real software. In order to solve such problems, we propose a self-adaptive software framework with runtime verification of the transition system. The proposed approach consists of two phases: a pre-computing phase that is responsible for the design and extraction of the transition system model for runtime performance, and a runtime phase that is responsible for monitoring and analyzing the designed transition model at runtime. We performed an empirical evaluation to demonstrate the excellent performance of our proposed approach.

2. RELATED WORK

In this section, we introduce several previous studies related to model checking on self-adaptive software and discuss their differences to the proposed approach. A study used model checking for verification of software modeling. It used model checking to evaluate resilience [7]. This study describes software and its requirements in terms of a model and computational tree logic (CTL). The model and CTL equation are used to evaluate the resilience after the adaptive activity ends. This study used model checking with its characteristics. Another study describes self-adaptive software in the form of multiple levels by using a state model [9]. This study classifies adaptation as structural and behavioral, and suggests that a system is described by state models. State models are located at different levels; therefore, lower-level models are verified at an upper-level model. Further, model checking is used to verify the models. These studies suitably apply model checking and state machine in self-adaptive software design and evaluation. However, they are optimized during pre-processing and post-processing; therefore, these approaches are not suitable for runtime verification.

Probabilistic verification has also been used for self-adaptive software at runtime [5-6]. The probabilistic model is pre-computed and translated as functions expressions. The final step in this approach is the generation of a verification condition set; this condition set is efficiently evaluated at runtime when changes occur. Thus, this approach supports sensitivity analysis. However, we assume that the probabilistic approach has inherent problems because the environment condition is not predictable in self-adaptive software at runtime. Therefore, we propose an environment-condition-based transition system model for the design and verification of a self-adaptive software. Our proposed approach attempts to solve the problems in previous related studies. The challenges addressed are as follows: verification at runtime, avoidance of the potential state-explosion problem.

3. PROPOSED APPROACH

3.1 Overview

We propose an approach for the design and verification of a transition system model for self-adaptive software. Self-adaptive software is described as a transition system model in order to integrate it with traditional verification and validation techniques (e.g., model checking). Therefore, we assume that self-adaptive software is designed as transition systems, and transitions of a designed model can be monitored. In addition, we applied model-checking theories (e.g., reachability) to verify self-adaptive software. Figure 1 shows an overview of the proposed approach, which consists of two phases: pre-computing phase and runtime phase.

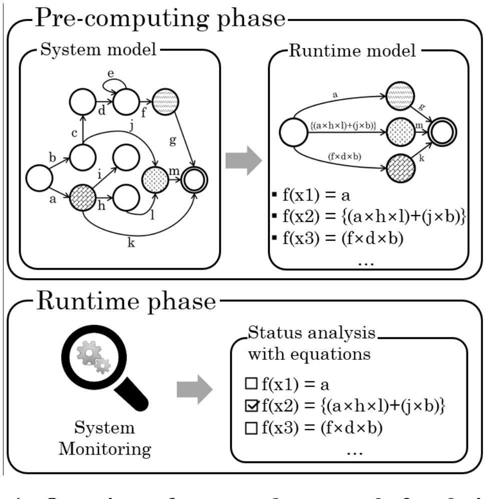

Figure 1: Overview of proposed approach for design and verification of self-adaptive software

The pre-computing phase models the system according to the traditional transition system model and abstracts the designed transition system into equation forms; then, the runtime transition model is reconstructed. The methods to design the model and extract the equations are described in Sections 3.2 and 3.3, respectively. The runtime phase monitors the system and analyzes the system based on equations that are extracted in the pre-computing phase. In the runtime phase, the system monitors the transitions of the designed transition model and can analyze only those equations that consist of transitions.

3.2 Transition System Model for Runtime Verification

We assume that a self-adaptive software is described as a transition system. In order to describe self-adaptive software, we apply the traditional transition system model [4]. The transition system is a tuple (S, Act, →, AP, L), where

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a set of transition,
- $s_0 \subseteq S$, and it is the initial state,
- AP is a set of atomic propositions, and
- $L: S \rightarrow 2^{AP}$ is a labeling function.

We assume that only one initial state exists, and transition values are described as Boolean or number values (i.e., 0 or 1). Further, the transition value can be monitored at runtime. However, this transition system is not suitable for runtime verification owing to a limitation; therefore, we propose a transition system for the verification of self-adaptive software at runtime by using the designed transition system. The proposed transition system is named Runtime Transition System (RTS).

RTS is a tuple (S, Act, →, AP, L), where

- S is a set of states,
- Act is a set of actions,
- States are classified into two types, $\{S_{normal}, S_{reach}\}$,
- $s_0 \subseteq S$, and it is the initial state,
- → is a set of transitions, and it is classified into two types $\{\rightarrow_{normal}, \rightarrow_{reach}\}$,
- $\rightarrow_{normal} \subseteq S_{normal} \times Act \times S_{normal}$ is a transition relation,
- $\rightarrow_{reach} \subseteq s_0 \times Act \times S_{reach}$ is a transition relation, and it is represented as an equation,
- AP is a set of atomic propositions, and
- $L: S \rightarrow 2^{AP}$ is a labeling function.

RTS consists of two types of states and transitions: S_{normal} is a normal state that does not impact self-adaptation, and normal transition (i.e., $\rightarrow_{normal}$) indicates the translation between normal states; S_{reach} is a set of states for which the software requirement is not satisfied in the pre-designed system model, and hence, if the software reaches this state, it could lead to abnormal termination or the occurrence of an error. Therefore, the reachable paths from the initial state (i.e., s_0) to each state of S_{reach} must be verified at runtime in a self-adaptive software environment. As described in Section 3.1, RTS is extracted with a pre-designed system model, and reachable transition (i.e., $\rightarrow_{reach}$) indicates all possible paths from the initial state to each S_{reach} state. By using the semantics of temporal modalities [4], the reachable transition for i is given below:

$$\rightarrow_{reach}(i) = \cup \{ \exists \Diamond S_{reach}(i) \} \tag{1}$$

"$\exists \Diamond S_{reach}(i)$" indicates that there exists a path that eventually reaches the ith state of S_{reach}. Therefore, "$\rightarrow_{reach}(i)$" is the union of the reachable paths to the ith state. Section 3.3 describes the method to extract a reachable transition from the pre-designed transition system and the method to translate the equations.

3.3 Extract Reachability from Transition System for Runtime Verification

In this study, we modified the state elimination algorithm [11] to extract the reachability of the transition system model. The method for reachability starts from S_{reach} states to initial state to avoid iteration problems. We describe the abstracting algorithm by using a simple transition system model (i.e., Figure 2).

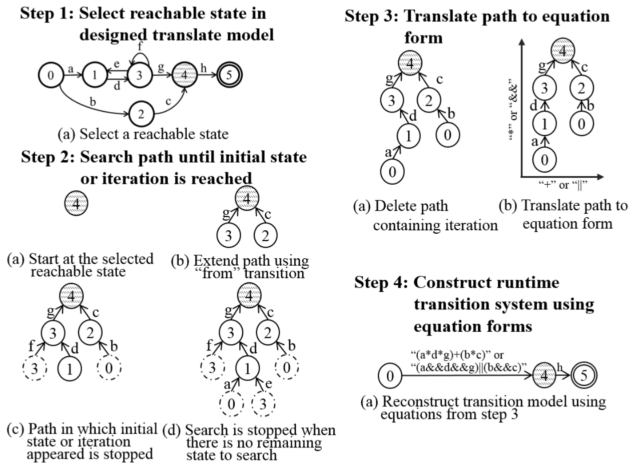
Step 1: Select reachable state in designed translate model
(a) Select a reachable state
Step 2: Search path until initial state or iteration is reached
(a) Start at the selected reachable state
(b) Extend path using "from" transition
(c) Path in which initial state or iteration appeared is stopped
(d) Search is stopped when there is no remaining state to search
Step 3: Translate path to equation form
"*" or "&&"
"+" or "||"
(a) Delete path containing iteration
(b) Translate path to equation form
Step 4: Construct runtime transition system using equation forms
"(a*d*g)+(b*c)" or "(a&&d&&g)||(b&&c)"
(a) Reconstruct transition model using equations from step 3

Figure 2: Process for extraction of reachability equation from transition model

RTS is extracted from the designed transition system in four steps. First, the reachable states are selected. As mentioned in Section 3.2, the reachable state indicates that the software requirement is not satisfied or contains potential errors. In the example, we select state "4" as a reachable state. After a reachable state is selected, path search is performed. The search starts from the reachable state; in the example, state "4" is the starting point for the path search (i.e., step 2-a in Figure 2). Then, the search process extends the path by using the "from" transition. In Figure 2, state "4" is translated from states "3" and "2"; therefore, the path is extended using these states (i.e., step 2-b). Next, the search process extends the path iteratively by using the "from" transition until the initial state or iteration state is reached. The search is stopped at the initial state because reaching the initial state indicates that a reachable path has been found. In addition, the search is stopped when an iteration appears because an infinite loop could be generated, thus leading to one of the problems in model checking (i.e., state explosion). In step 2-c, state "3" has an iterative "from" transition; therefore, the path search is stopped at state "3". Further, state "2" has a transition from initial state "0"; therefore, the path search is stopped at state "0". In the example (i.e., step 2-c), path searching is continued for state "1". The path-search process is continued until there is no state for extension. In step 2-d, there is no state for extension because all the states are connected with the initial state or iteration state.

After the path search, the paths are translated to equation form. In this process (i.e., step 3-a), an iterative path is deleted to satisfy the reachability definition described in section 3.1. Then, each path is translated to the equation form (i.e., step 3-b). Linear paths are converted to "multiplication" operations or "and (&&)" operations because a linear path indicates that if one transition value is unable to reach the next state, the remaining states are also unreachable. For example, in step 3-b, if transition "d" is unable to reach the next state, then a path containing "d" (e.g., 0→1→3→4) cannot reach state "4". In addition, parallel paths are converted to "plus" operations or "or (||)" operations because even if one path cannot reach $S_{reach}(i)$, an alternative path is possible. In step 3-b, if one path (e.g., 0→1→3→4) is impossible, another path is possible (e.g., 0→2→4). After paths are translated to the equation form, a pre-designed transition model is reconstructed as an RTS by using the equations extracted in step 3. In addition, the extracted equations are calculated for the self-adaptive software at runtime in order to verify the system status. We implemented a prototype of the proposed method by using Java SE (version 1.8) to ensure compatibility of various devices. A detailed description of the implementation is outside the scope of this manuscript; therefore, we have omitted the implementation details.

4. EMPIRICAL EVALUATION

This section discusses a set of experiments for empirical evaluation. For the experiments, we generated a data set that would yield the worst performance with the proposed method. With regard to the connectivity of the transition system, all generated transition system has at least one in-transition and one out-transition, except the start-state and end-state. The end-state has only one in-transition, and we assume that the abstracted state is the end-state. Therefore, all transitions must be searched to obtain the RTS. The test set consists of 20 finite state machines for each state size, and we calculate the mean of each data set. The experiment measures the time required to abstract the transition system. The results obtained from the increasing state sizes are shown in Figure 3. In the experimental setup, the hardware consists of Intel® Core™ i7-2620M CPU (2.7 GHz) and 8 GB memory. The software environment consists of Windows 7 Professional and Java 1.8.

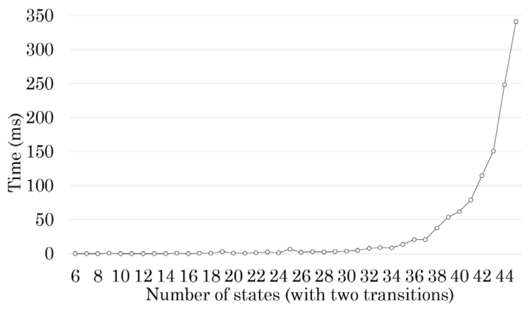

Figure 3: Results of equation extraction with increasing state number

We observe that as the state size increases, the computation time required increases. When the state size is 45, only 341 ms are required. We consider that the pre-computing phase is executed before the runtime phase. Therefore, runtime computation time does not affect the execution of reachability verification at runtime.

After the experiments for the pre-computing phase, we performed experiments on runtime performance by using data sets from previous experiments and RTSs. We obtained equation results by randomly generating transition values (i.e., from 0 to 1). We compared the proposed method and the model-checking tools. We selected NuSMV[1] and CadenceSMV[2] as the comparison tools. NuSMV and CadenceSMV, which is one of the powerful tools in model verification, is a symbolic model checker; further, these tools are open-source. In the experiments, we measured the time required to achieve "reachability" from the start to the end-state by using NuSMV and CadenceSMV. Based on intuitive semantics of temporal modalities [4], "reachability" is denoted as:

$$\varphi = \exists \Diamond \text{state}_{\text{end}} \qquad (2)$$

Equation φ (i.e., equation 2) indicates that there exists a path that eventually reaches the end-state. In order to calculate φ, NuSMV and CadenceSMV can obtain only one path from the start-state to the end-state. The results of runtime performance comparison are shown in Figure 4.

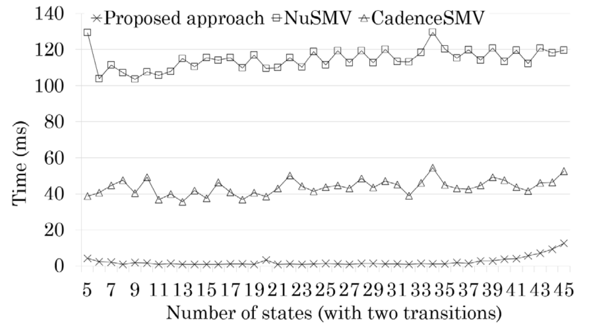

Figure 4: Results of runtime comparison with increasing states

The results show that the proposed method performs better than the path-finding (i.e., reachability) of NuSMV and Cadence SMV. NuSMV and CadenceSMV differ from the proposed method when the node size increases. NuSMV and CadenceSMV exhibit monotonic change with increasing nodes. These tools terminate model checking when they find a path that reaches the specific state. In equation φ, these tools end their model checking when a path reaches an end-state. Therefore, these model checkers provide only a single path to reach an end-state; however, the proposed method can consider various scenarios to reach an end-state. Further, the proposed method is faster than the other tools. The proposed method has a pre-compile process that converts the pre-designed transition system to RTS. Although the proposed method includes a pre-compile process, it has the advantage of saving time at runtime by considering several model-checking cases.

5. CONCLUSION

In this manuscript, we have proposed a runtime verification approach for self-adaptive software by using reachability. The proposed approach has two phases. In the first phase, the process is translated from a pre-designed translation system to a runtime translating system model. In this phase, the set of states containing a potential error is selected, and the path to reach that set of states is calculated. The calculated reachable paths are translated into the equation form. In the second phase, the transition values described in the transition model are monitored, and then, the equations extracted in the previous phase are analyzed. We performed experiments with other model-checking tools; the experiments demonstrate that the proposed approach is suitable for runtime verification.

In this study, we demonstrate the suitability of the proposed approach; however, this approach must be integrated in the self-adaptive lifecycle. Therefore, we plan to extend the proposed approach with a MAPE (monitoring, analyzing, planning and executing)-loop in an IoT-based environment. Further, we will extend our method to include a logical model such as linear temporal logic (LTL) or computational tree logic (CTL) in order to provide a specific description of software requirements.

6. ACKNOWLEDGMENTS

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 (NRF 2012M3C4A7033346).

7. REFERENCES

[1] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.
[2] G. Tamura, N. M. Villegas, et al. Towards practical¨ runtime verification and validation of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 108–132. Springer, 2013.
[3] R. De Lemos, H. Giese, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.
[4] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of model checking*. MIT press, 2008.
[5] A. Filieri and G. Tamburrelli. Probabilistic verification at runtime for self-adaptive systems. In *Assurances for Self-Adaptive Systems*, pages 30–59. Springer, 2013.
[6] A. Filieri, G. Tamburrelli, and C. Ghezzi. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. *IEEE Transactions on Software Engineering*, 42(1):75–99, 2016.
[7] J. Cámara, and R. de Lemos. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 53–62. IEEE Press, 2012.
[8] K. Johnson, R. Calinescu, and S. Kikuchi. An incremental verification framework for component-based software systems. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pages 33–42. ACM, 2013.
[9] L. Tesei, E. Merelli, and N. Paoletti. Multiple levels in self-adaptive complex systems: A state-based approach. In *Proceedings of the European Conference on Complex Systems 2012*, pages 1033–1050. Springer, 2013.
[10] 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*, pages 199–210. ACM, 2014.
[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley. 2007

[1] http://nusmv.fbk.eu/
[2] http://www.kenmcmil.com/smv.html