Expansion Mechanism for Runtime Verification of Self-adaptive Systems

Masaya Fujimoto, Hiroyuki Nakagawa, Tatsuhiro Tsuchiya
Graduate School of Information Science and Technology, Osaka University
Email: {m-fujimoto, nakagawa, t-tutiya}@ist.osaka-u.ac.jp

***Abstract*—Self-adaptive systems can adapt to environmental changes by modifying their behavior and require runtime verification after adaptation. More efficient verification mechanisms are required because verification mechanisms such as model checking are computationally and memory intensive. A possible method is to generate expressions for model checking at design time and execute such expressions at runtime. Our previous work proposed a caching mechanism and parameterization to improve the expression generation method. In this study, we improve our previous work by generating expressions using Laplace expansion. This method expands the probabilistic model at the points where it is different from the design model and brings the model closer to a model in a cache for generating expressions. We also propose a method to generate candidate metrices to increase the number of cached matrices and improve the cache hit ratio. We conducted experiments with three types of changes, that is, adding, changing, and deleting states. We observed that our approach is effective when the model's states are added or changed.**

Keywords- requirements, self-adaptive system, runtime verification.

I. INTRODUCTION

Software systems are increasingly being used by more people and should be highly reliable regardless of changing operating environments. Self-adaptive systems [1,2,3,4] are systems that can operate stably in changeable environments. A self-adaptive system can restructure itself in response to changes in the external environment, making it easy to change and manage the system. However, a self-adaptive system requires verification at runtime to ensure that the modified system meets system requirements. We focus on the reachability property, which states that a target state can be eventually reached from an initial state. The state space is represented as a discrete-time Markov chain (DTMC) [5] model. This is because many properties useful in software development can be reduced to reachability. Filieri et al. [6,7] proposed a method that places the state transitions that are expected to be changed as variables in advance. This method generates a set of verification expressions that include the variable at design time and assigns obtained parameters at runtime to perform verification. This allows for faster verification. However, Filieri's method generates a verification expression for system behavior at design time; therefore, it is not possible to perform verification quickly when the behavior changes significantly after adaptation. To address this problem, our previous work [8,9] proposed a method that uses a caching mechanism: Intermediate formulas obtained during the generation of verification formulas are stored in a cache. When the system state changes significantly and recalculation is necessary, the results of the intermediate formulas are reused from the cache, thereby reducing the computation time during system execution. Furthermore, by generating predicted models after adaptation and storing them in the cache as well, the cache hit ratio is improved, further decreasing computation time.

DOI reference number: 10.18293/SEKE2023-223

In this study, we attempt to further speed up runtime verification time by improving the caching method in our previous study. Previously, the cache size increased along with the size of the predicted model, resulting in slower execution times. In this study, we use model modification information and perform Laplace expansion from the point where the model state changes when re-generation is necessary. We also generate candidate matrices to improve the cache hit ratio and reduce the calculations done in generating expressions.

The experiment results shows that this approach is faster than other methods in specified situation, such as adding states and changing states.

II. RELATED WORK

Previous studies have developed the approach of model checking for runtime verification. For example, [10] proposed a fast parametric model checking (fPMC) approach that generates an abstract model, which represents multiple states with a single state. Thus, even if the model size increases, this method can reduce the computational time. Furthermore, [11] proposed an incremental quantitative verification method for probabilistic models, which re-uses results from previous runtime verification to accelerate the process. The key in this approach is to use a decomposition of the model into its strongly connected

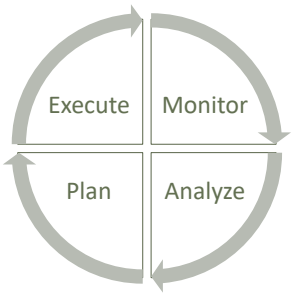

Fig 1. MAPE feedback loop mechanism.

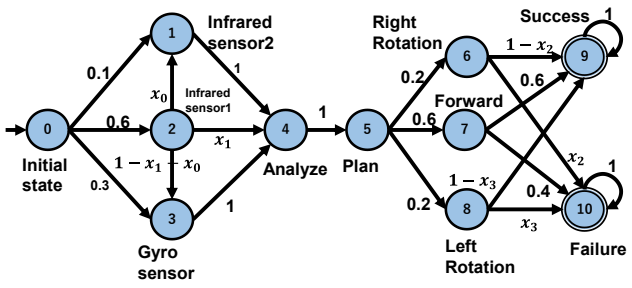

Fig. 2. An example of DTMC model verification: a cleaning robot.

components (SCCs). This method also uses the structure of models and requires the analysis of the change impact on the model before the previous verification. Ref. [12] investigated three techniques, namely caching, lookahead and nearly-optimal reconfiguration. While this technique assumes that the verification is continuously executed, our technique assumes that the verification is divided into at runtime and at design time. A technique of caching is the basis of our approach. A technique of lookahead uses spare CPU to pre-verify stochastic models, which are expected to arise in the future. A technique of nearly-optimal reconfiguration terminates runtime verification as soon as a system configuration satisfies some condition. This paper shows that these techniques can lead to significant reductions in runtime verification.

III. BACKGROUND

In this section we explain self-adaptive systems and the DTMC method of model verification, which is widely used to model the reliability of software systems. In our study, we assume that the system states are represented as models, which enable easy and efficient verification of the reliability of systems in terms of non-functional properties, such as memory consumption and the computational cost.

A. Self-adaptive systems and model verification

The primary mechanism for self-adaptive systems is the monitor-analyze-plan-execute over a shared Knowledge (MAPE-K) feedback control loop [13], which repeats the four steps: monitoring, analyze, plan, and execute (Fig. 1). At runtime, a self-adaptive system monitors its external environment and analyzes information obtained from monitoring. If the system state violates the requirements, the system plans a new behavior that meets requirements and updates itself. This mechanism enables automatic adaptation to the environment. A self-adaptive system requires efficient verification [14] to meet requirements and update itself over time.

B. Discrete-time Markov Chcain Model

A DTMC model is defined as state transition augmented with probabilities that meet the Markov process requirement that future states depend only on the current states without depending on previous states. The elements of a DTMC model are as follows:

- S is a finite set of states

$$\begin{pmatrix}
0 & 0.1 & 0.6 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & x_0 & 1-x_0-x_1 & x_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0.6 & 0.2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1-x_2 & x_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0.4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1-x_3 & x_3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Fig. 3. The transition matrix of Fig. 2.

$$\begin{array}{c}
\begin{matrix} Q & & & & & & & & & R & \end{matrix} \\
\left(\begin{array}{ccccccccccc}
0 & 0.1 & 0.6 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & x_0 & 1-x_0-x_1 & x_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0.6 & 0.2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1-x_2 & x_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0.4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1-x_3 & x_3 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right) \\
\begin{matrix} O & & & & & & & & & I & \end{matrix}
\end{array}$$

Fig. 4. Sub-matrices Q, R, O and I of the DTMC model for the cleaning robot illustrated in Fig. 2.

- $S_0 (\subseteq S)$ is a set of initial states
- P: S × $S \rightarrow [0,1]$ is a transition matrix representing the transition probability between states

A DTMC model has two types of states. The first is an absorbing state, which has transition probability of 1 to itself, while the second is a transient state which has a transition to other states. In this study, the state transition probability is represented as a real value [0, 1] and variables.

Fig.2 shows a model of how a cleaning robot acts. This model analyzes the information obtained by the sensor and decides actions based on the information. The model transitions to either a failure or success state. The circles in Fig.2 denote states and each arrow represents a transition from a state to a next state. The number on each arrow stem denotes the probability of a state transition. The probability variable is a parameter obtained through execution or expected to change. The system starts at an initial state 0 and transitions to states 1-3, based on information acquired by the sensors. External information is acquired via infrared sensor 1. If infrared sensor 1 has a problem, it transitions to states 1, or 3. In state 4, the system analyzes the information obtained. In state 5, it plans response actions based on the analysis, and transitions to states 6-8. States 6-8 indicate the robot motions; if a movement action is performed without any problems, a transition is made to state 9, which denotes success. Conversely, if the corrective action cannot be performed owing to obstacles, a transition is made to state 10, which denotes a failure state.

A DTMC model can be represented by an adjacency matrix. Fig.3 shows a matrix representation of the example in Fig. 2. In such a matrix, row i, column j represents the probability of transition from state i to state j.

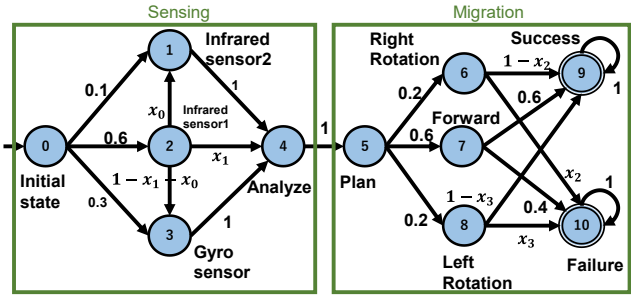

Fig. 5. An example of grouping of states.

C. Runtime verification using DTMC model

In this section, we explain how to calculate state transition probabilities for model checking from a DTMC model and system requirements. A DTMC model with absorbing states can be represented by the following four matrices.

$$P = \begin{pmatrix} Q & R \\ O & I \end{pmatrix}$$

The matrix Q is a matrix of probabilities of transitioning from a transient to transient state; the matrix R is a matrix of probabilities of transitioning from a transient to an absorbing state, and the matrix I is a matrix of probabilities of transitioning from an absorbing to absorbing state. The matrix I is an identity matrix because the transition probability to itself is 1. The matrix O represents the transition probability from an absorbing to a transient state, which is always zero because an absorbing state has only transitions to itself. Hence, matrix O can be expressed as a zero matrix, as shown in Fig. 4.

Reachability in DTMC model can be expressed by the probability operator $P_{\bowtie p}(l)$, where l is a path formula. $\bowtie$ denotes the comparison operator, such as $<$, $\leq$, $>$, and $\geq$, and p is a threshold of the probability that is defined by the requirements. $P_{\bowtie p}(l)$ represents whether the probability meets $\bowtie p$ under the condition l. We verify whether the probability of reaching an absorbing state satisfies $\bowtie p$.The following section describes how to obtain the transition probabilities for verifying reachability. To verify reachability in a DTMC, we consider the transition probability from a transient to transient state. If Q denotes the transition probability from a transient state, the transition probability after the first two transitions can be expressed as Q^2, which is the product of the first and second transition probabilities. The probability in some steps can be calculated in same way as follows.

$$\mathrm{N} = \mathrm{I} + Q^1 + Q^2 + Q^3 + \cdots = \sum_{k=0}^{\infty} Q^k$$

Because matrix N is an infinite series of matrix Q, matrix N can be taken as the inverse matrix of matrix $(I - Q)$. Next, given that matrix N is the transition probability from a transient to transient state, the reachability can be obtained with the following equation.

$$\mathrm{B} = \mathrm{N} \times R$$

Reachability b_{ik} from an initial state S_i to an absorbing state S_j can be calculated as follows.

$$n_{ij} = \frac{1}{det(W)} \cdot \alpha_{ji}(W)$$

$$b_{ik} = \sum_{x \in 0..t-1} n_{ix} \cdot r_{xi} = \frac{1}{det(W)} \alpha_{xi}(W) \cdot r_{xj}$$

The calculation of b_{ik} requires the calculation of determinants. The determinant is calculated by Laplace expansion and LU-decomposition.

D. Generating runtime verification expressions

The calculation of determinants is computationally intensive but must be performed at runtime. Therefore, Filieri et al. [5,6] proposed a method by performing some of the calculations required for model checking at design time.

The method consists of two processes: precomputation at design time and verification at runtime. First, precomputation parameters that can only be obtained or may change at runtime are placed as variables, and verification expressions are generated. At runtime, the desired transition probabilities are calculated by substituting parameters into the verification equation to determine whether the requirements are met. This allows for fast model checking even if some of the transition probabilities have unknown parameters. The transition probabilities calculated by this method from initial state 0 to absorption state 10 in Fig. 3 are as follows:

$$b_{010(x_0,x_1,x_2,x_3)} = \frac{0.02(x_0 + x_1)(x_2 + x_3) + 0.024(x_0 + x_1)}{-x_0 - x_1}$$

At runtime, the reachability property can be obtained by substituting parameter values obtained from sensors and other sources into this expression.

In this method, LU-decomposition is not possible when the matrix includes variables. Therefore, Laplace expansion is performed first; then the variables are removed from the matrix to enable LU-decomposition. This allows for shorter computation times than would have been with Laplace decomposition alone. We denote the size of Q matrix as t, the average transition number as τ, and the number of rows including variables as c. To calculate b_{ik}, the calculation of t determinants that is $(t-1) \times (t-1)$ sizes of sub-matrices using Laplace expansion is given by $O(t^3)$. In the case of calculation of determinants with variables, the row including variables is expanded and requires τ^c determinants. The expanded matrices are then calculated by LU-decomposition because the matrix has no variables. The calculation of a runtime verification formula is as follows:

$$O(\tau^c \cdot (t-c)^3) \sim O(\tau^c \cdot t^3)$$

E. Caching mechanism and grouping of states

In this section we describe the caching mechanism in the proposed runtime verification reduction method. Filieri's method cannot use the generated expression when the system has changed significantly, such as if states have been added and deleted, and needs to re-generate the expression. The re-

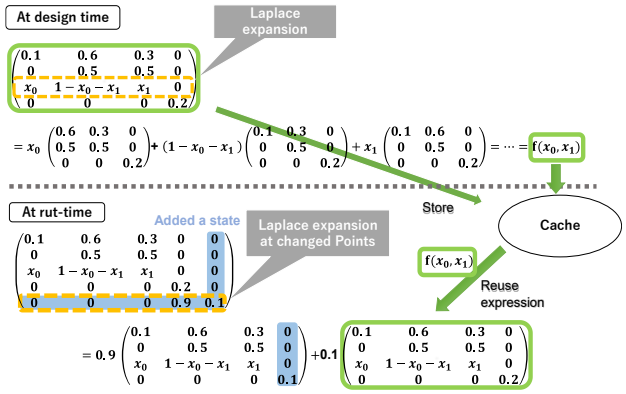

Fig. 6. An example of the proposed method, which uses on Laplace expansion.

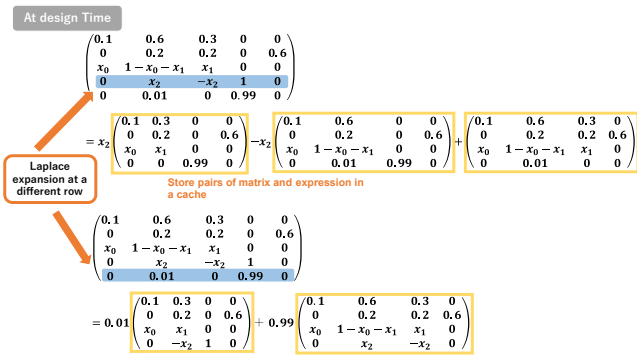

Fig. 7. An example of generation of candidate matrices.

calculation of the generated verification formula at design time is time-consuming and affects system performance and execution. To address this problem, a caching mechanism is used. This caching mechanism stores pairs of matrices and the intermediate expressions obtained during the generation of verification formulas and reuses an expression if the matrices match. This is because the changes in self-adaptive systems are partial, and most of the models are similar.

To reduce computational time, this approach generates candidate models and stores their matrix pairs as well. This can enable an early match of a matrix during calculation. Only matrices including the variable are stored in a cache, because a matrix without variables can be efficiently calculated by LU-decomposition. When the size of model matched is large, a correspondingly large computational time improvement is gained from using a cache. The converse is the case when the matched model size is small.

The caching mechanism has a drawback in that the size of the system model increases as the number of stored matrices in a cache becomes large. To solve this problem, in our previous study, similar states are grouped, and the transition of states is limited within the same group. The assumption is that processes can typically be grouped by functions. Hence, processes in the same group are completed in one function and adding of states is restricted to the same group. This can reduce the number of candidates.

Fig.5 represents the grouping of states example in Fig.2. States 0-4 belong to sensing group and states 5-10 are in migration group.

IV. Efficinet Laplace Expansion for Caching

The cache mechanism uses intermediate formulas to generate expressions efficiently. As model size increases, cache size also increases. An increasing cache size makes it difficult to perform by the robot, which has restriction of memory usage. Therefore, to improve hit ratio without increasing the cache size, we focus on Laplace expansion. In Laplace expansion, for a square matrix of order n, row I is chosen arbitrarily and coefficient A_{ij} is calculated for each component of the i th row, so that the resulting expansion formula matches the determinant of A. The equation is expressed as follows:

$$\det(\mathrm{A}) = \sum_{\mathrm{j}=1}^{\mathrm{n}} (-1)^{i+j} a_{ij} \cdot \det(\mathrm{A_{ij}})$$

Laplace expansion can be applied to column j in a similar process by calculating the coefficient $\mathrm{A_{ij}}$ for each component in column j to obtain the determinant of A.

To improve the cache hit ratio, we store changes in the model at design time. At execute time, we perform Laplace expansion on the changed points to remove them. Compared to other methods, the matrix does not contain the changed points after adaptation; thus, it is more likely to match the matrices stored in the cache.

Fig.6 shows an example of how Laplace expansion is performed, and the resulting generated intermediate expression and its matrix are stored in the cache. At runtime, if the matrix does not include any variables, the matrix is calculated by LU-decomposition because LU-decomposition is faster than Laplace expansion. If the matrix includes any variables, the system acquires added rows as model change information and performs Laplace expansion on the rows added during the calculation of determinants. A search is performed on the obtained matrices, and if a matching matrix exists, the matrix is replaced with the expressions corresponding to the matrix.

V. Generating Candidate Matrices

To improve the cache hit ratio, we expand the matrix at design time in different rows, and intermediate expressions are stored in a cache. This increases the number of matrices stored in the cache and improves the cache hit ratio. Reducing the computational time at runtime depends on replacing the calculation of determinants with expressions stored at design time. This requires more matrices corresponds with partial changes.

Fig.7 shows an example of generation of candidate matrices. The third row of the top matrix in the figure is expanded. The expanded matrix and the verification expression from the calculation are stored in a cache. Laplace expansion is done on the fourth row of the bottom matrix in the figure. The resulting matrix is different from the one obtained by Laplace expansion on the third row. The intermediate formulas for these matrices

TAB. I RESULTS OF EXPERIMENTS ON ADDING STATES WHEN THE SIZE OF MODEL IS CHANGED

	Execute Time [ms]				Cache Hit Ratio				Cache Size [KB]			
	Baseline Method 1	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*	*Baseline Method 1*	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*	*Baseline Method 1*	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*
10	13.349	8.824	6.468	4.424	0	0.07710	0.3029	0.48564	0	105.274	3.744	40.051
15	192.841	133.019	57.737	11.470	0	0.01772	0.22672	0.59066	0	999.584	13.402	324.704
20	677.729	662.884	255.478	20.201	0	0.00572	0.18992	0.61975	0	3091.104	21.990	1037.523
25	2338.353	2278.914	795.951	50.123	0	0.00142	0.12013	0.61611	0	6527.738	32.525	2330.170

TAB. II RESULTS OF EXPERIMENTS ON CHANGING STATES WHEN THE SIZE OF MODEL IS CHANGED

	Execute Time [ms]				Cache Hit Ratio				Cache Size [KB]			
	Baseline Method 1	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*	*Baseline Method 1*	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*	*Baseline Method 1*	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*
10	13.752	11.17	19.282	6.419	0	0.11517	0.09475	0.10473	0	100.186	7.2	64.934
15	389.014	270.952	355.673	157.602	0	0.13521	0.11095	0.08238	0	3007.06	65.741	1037.952
20	3451.744	3472.968	5005.558	2631.507	0	0.01082	0.00766	0.05104	0	17082.573	208.378	6373.037
25	15957.526	17634.146	21003.851	12243.737	0	0.00745	0.00488	0.04352	0	50554.291	433.862	20658.054

TAB. III RESULTS OF EXPERIMENTS ON DELETING STATES WHEN THE SIZE OF MODEL IS CHANGED

	Execute Time [ms]				Cache Hit Ratio				Cache Size [KB]			
	Baseline Method 1	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*	*Baseline Method 1*	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*	*Baseline Method 1*	*Baseline Method 2*	*Proposed Method 1*	*Proposed Method 2*
10	1.792	2.412	1.371	1.986	0	0.18576	0.29313	0.46791	0	92.192	6.579	62.15
15	26.777	19.018	35.371	11.694	0	0.05338	0.0578	0.27756	0	2005.037	42.912	813.434
20	312.608	328.058	630.843	468.733	0	0.00923	0.00675	0.21231	0	13078.673	238.944	7049.606
25	1333.708	1695.101	3427.358	3194.640	0	0.00122	0.0055	0.09127	0	35126.054	297.363	15790.061

are stored in the cache as well to efficiently use the model at design time. This increases the number of matrices stored in the cache.

VI. EXPERIMENTS

The experiments compare the calculation time and cache size of the proposed method with those of other methods. In addition, the cache hit ratio is compared; that is the number of matches with the matrix stored in the cache at runtime divided by the number of cache searches. All the programs used in the experiments were implemented in Java. The experiments were conducted using the following methodology.

- State size ranges from 10 to 25 states; the increment is by five states.
- Transition from one state to another is randomly generated within the same group.
- The number of trails is 10, and the average computation time over the 10 trials is used.
- The computation time and cache size required to obtain the probability of transition to absorbing state are measured.

Methods to be compared.

- Baseline method 1: Filieri's method
- Baseline method 2: Intermediate generative formulas using a cache + candidate model.
- Proposed Method 1: Using Laplace expansion.
- Proposed Method 2: Proposed method 1 + generating of candidate matrices.

Baseline method 1 is the method by Filieri et al. Baseline method 2 uses a cache, generates candidate models at design time, and stores them in the cache. Proposed method 1 uses efficient Laplace expansion. Proposed method 2 performs Laplace expansion at different points at stores the matrices in the cache. We conducted experiments on three types of changes, namely adding states, changing states, and deleting states. The first experiment adds a design model to one state whose transition probability is restricted to the same group. Second experiment changes transition probabilities in one row. The last experiment randomly deletes one state in the design model. We conducted this experiment on a Mac equipped with a 7th generation Core m3(1.2GHz), 8.0 GB RAM, and Java program running on Eclipse.

Tab. I shows the results when the number of states is changed by adding states. The execution times of the proposed methods 1 and 2 show that they can generate runtime verification expressions in less time than other methods. Baseline method 1 does not use a cache; hence, cache hit ratio is 0. Among the other methods, the proposed method 2 has a high cache ratio of 0.61 for 25 states. The proposed method 2 has a smaller cache size than the proposed method 1. Tab. II shows the results when the number of states is changed, and states are partially changed. The execution time of the proposed method 2 is faster than other methods in generating a runtime verification expression, while the proposed method 1 is slower than other methods when the number of states is 25. Despite adding states as well, the cache size and cache hit ratio of baseline method 1 is 0. Tab. III shows the results when the number of states is changed by deleting

states. The execution time of method 1 is the fastest and proposed method 1 is the slowest in these experiments.

VII. Discussion

A. Experiments on adding states

In the experiments on adding states, proposed method 2 is faster than other methods in generating a runtime verification expression. Efficient Laplace expansion enhances the cache hit ratio and leads to reduction in computational time by replacing the calculation of determinants with expressions. In particular, the cache hit ratio of proposed method 2 is stable owing to large size of matrices in the early phase, whereas the cache hit ratio of the other methods decreases.

B. Experiments on changing states

Tab. II shows the execution time for "changing states" experiments. The execution time of the proposed method 2 is less than those of other methods; however, the proposed method 1 takes the longest time to generate an expression. This is because the number of intermediate expressions stored with proposed method 1 is smaller than the number stored with proposed method 2. The cache of proposed method 1 does not include the matrices in cases of partial model changes. This leads to waste of cache search and increases computational time. Compared to adding states, as the states increase, the cache hit ratio of proposed method 2 is lower because of having to match a small number of matrices.

C. Experiments on deleting states

Tab. III shows the results for deletion of state. Our proposed method is not effective in deletion of states, method 1 is the fastest, and the proposed method 2 is the slowest of the methods. Because proposed methods 1 and 2 do not have enough matrices (which corresponds with deleting states), search misses increase execute time. The cache hit ratio of the proposed method 2 is higher than those of the other methods, because cache hit locally is much higher. We should consider improvement to increase the number of matrices in deleting states situations.

These experiments demonstrate that our approach is effective in adding and changing states but is not effective in deleting states. Additionally, we found that the size of the matrices matched affects the execution time. In future work we will devise a method that matches as large a size of model as possible. We would decrease the cache size to increase the applicability of this method.

VIII. Conclusion

In this study we described a runtime verification mechanism for self-adaptive system. We propose the method using efficient Laplace expansion for caching. We also generate candidate matrices to increase storage of intermediate runtime verification expressions when significant changes occur, such as adding, deleting, and changing of states. These approaches improved the cache hit ratio by expanding the changed points of the matrix and reduced the computational time for generating runtime verification expressions. This led to fast runtime verification. The proposed method, which generates candidate matrices and expands the changed points, is effective when the states can be grouped and the model undergoes changes, such as adding or changing states. A possible application of the proposed mechanism is dynamic web applications.

In future research, we aim to improve this method with experiments on deletion of states and partially changing states. Additionally, we will consider different adaptation patterns, such as adding more states and changing the number of variables to extend the applicability of the caching mechanism.

Acknowledgment

This work was supported by JSPS Grants-in-Aid for Scientific Research (No.20H04167).

References

[1] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, “Model evolution by runtime parameter adaptation,” Proceedings of the 31st International Conference on Software Engineering, pp.111–121, ICSE ’09, IEEE Computer Society, Washington, DC, USA, 2009.

[2] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, “Self-adaptive software needs quantitative verification at runtime,” Commun. ACM, vol.55, no.9, pp.69–77, Sept. 2012.

[3] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, “Dynamic qos management and optimization in service-based systems,” IEEE Transactions on Software Engineering, vol.37, no.3, pp.387–409, May 2011.

[4] J. Zhang and B.H.C. Cheng, “Model-based development of dynamically adaptive software,” Proceedings of the 28th International Conference on Software Engineering, pp.371–380, ICSE ’06, ACM, New York, NY, USA, 2006.

[5] C. Baier and J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.

[6] A. Filieri, C. Ghezzi, and G. Tamburrelli, “Run-time efficient probabilistic model checking,” Proceedings of the 33rd International Conference on Software Engineering,pp.341–350, ICSE ’11, ACM, New York, NY, USA, 2011.

[7] Filieri, G. Tamburrelli, and C. Ghezzi, “Supporting self-adaptation via quantitative verification and sensitivity analysis at run time,” IEEE Transactions on Software Engineering, vol.42, no.1, pp.75–99, Jan. 2016.

[8] H. Nakagawa, K. Ogawa, and T. Tsuchiya, “Caching strategies for run-time probabilistic model checking,” in Proc. of the 11th International Workshop on Models@run.time (MRT 2016), pp.18, Oct. 2016.

[9] H. Nakagawa, H. Toyama, T. Tsuchiya, Expression caching for runtime verification based on parameterized probabilistic models, Journal of Systems and Software 156 (2019) 300–311.

[10] X. Fang, R. Calinescu, S. Gerasimou and F. Alhwikem, “Fast Parametric Model Checking through Model Fragmentation,” 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, ES, 2021, pp. 835-846.

[11] M. Kwiatkowska, D. Parker and H. Qu, “Incremental quantitative verification for Markov decision processes,” 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), Hong Kong, China, 2011, pp. 359-370.

[12] Simos Gerasimou, Radu Calinescu, and Alec Banks. 2014. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014). Association for Computing Machinery, New York, NY, USA, 115–124.

[13] K. Goseva-Popstojanova and K. Trivedi, “Architecture-based approach to reliability assessment of software systems,” Performance Evaluation, vol.45, pp.179–204, 07, 2001.

[14] Taylor, H.E., Karlin, S., 1998. An Introduction to Stochastic Modeling. Academic Press.