Caching Strategies for Run-time Probabilistic Model Checking

Hiroyuki Nakagawa, Kento Ogawa, Tatsuhiro Tsuchiya
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
Email: {nakagawa, o-kento, t-tutiya}@ist.osaka-u.ac.jp

***Abstract*—For software systems that need to adapt to their environment at run-time, run-time verification is useful to guarantee the correctness of their behaviors. Probabilistic model checking using discrete time Markov chain (DTMC) model has been applied to implement run-time verification. A current existing approach provides an efficient run-time verification mechanism by pre-generating expressions for model checking at design time. In case that a system model is changed, the system is required to re-generate the expressions. In order to expand the applicability of the approach, we propose three strategies, caching, prediction, and reduction, for reducing computational time for re-generated expressions at run-time. We conduct preliminary experiments and demonstrate that our approach could expand the applicability of run-time verification by reducing the computational cost at run-time.**

I. Introduction

Software systems are often required to be able to maintain high performance and high reliability even though their environments are changeable or uncertain at design time. Traditional software systems are difficult to demonstrate the best performance or even run correctly under their changeable or uncertain environments. Constructing systems as self-adaptive systems [1], [2], which enable run-time adaptation, is an efficient way of dealing with such environments. Since self-adaptive systems modify their behaviors for adaptation, first, the systems should be able to autonomously control their behaviors. Model-based approaches [3], in particular, models@run.time [4] approach, which focuses on dynamically changeable models of the systems, provide a promising way of modifying behaviors at runtime. Second, the correctness of their behavior after adaptation should be verified. Such systems can benefit from a run-time verification mechanism [5], [6], [7] that checks whether a new behavior satisfies system requirements.

Unlike verification techniques at design-time, run-time verification techniques should be efficiently executed not to degrade system performance. Some studies on efficient verification for self-adaptive systems have been developed. Filieri et al [8] proposed an efficient run-time verification technique that divided a verification process into pre-computation at design time and value calculation at run-time. Pre-computation constructs expressions for verification from a system model described in discrete time Markov chain (DTMC) model [9], which associates state transitions with probabilities represented by fixed values or parameter variables, and requirements described in probabilistic computational tree logic (PCTL) at design time; value calculation assigns values that are measured by monitoring the environment to the variables in expressions at run-time. In such a verification process, the verification activity during run-time is only substituting the variables with values obtained by monitoring the environment. Although the process allows efficient run-time verification, this method assumes that the structure of the system model is not changed, that is, states or transitions are not added or deleted. If such changes happen, the system has to update (re-generate) the expression at run-time. This re-generation usually requires large computational time.

In expression generation, variables in DTMC models, which represent uncertain transition probabilities, lead to a large computational cost. DTMC models are represented as matrices, and the expression generation requires the computation of *determinants*. The computation of determinants is performed by using *Laplace expansion* and *LU-decomposition*. We previously evaluated the computational cost for generating expressions and concluded that the number of times of performing Laplace expansions considerably affects the computational cost [10].

In this study, we aim to expand the applicability of the efficient run-time verification technique by reducing the computational time for executing Laplace expansions. We reuse intermediate expressions obtained in pre-computation. Our approach is on the basis of *caching*. We store pairs of a partial matrix in DTMC and the corresponding intermediate expression as key-value pairs into cache. Moreover, to increase cache hit ratio and decrease cache size, we apply additional strategies, *prediction* and *reduction*.

We conduct preliminary experiments on randomly generated DTMC models. The experimental results demonstrate that our approach could expand the applicability of run-time verification by reducing the computational cost at run-time while keeping from rapidly increasing memory usage.

This paper is organized as follows: Section II describes theoretical background of our study. Section III explains our approach for an efficient run-time verification with three strategies. Section IV evaluates our approach from preliminary experimental results. Section V presents related work, and Section VI concludes this paper.

II. Theoretical Background

Since systems that need to adapt to their environment have to face uncertainty, we deal with the systems under

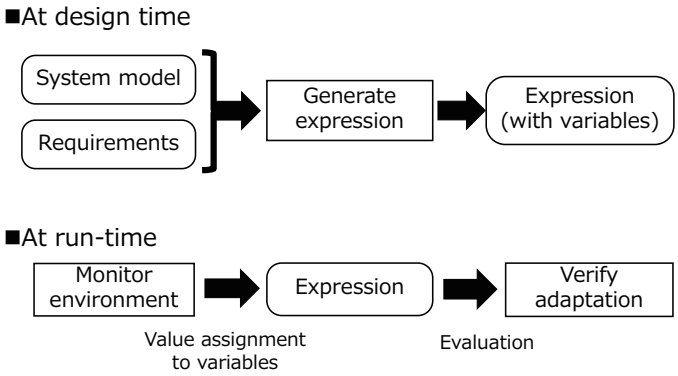

Fig. 1. Filieri's approach [8].

development as probabilistic systems. In many cases, behaviors of such systems are described in Discrete Time Markov Chain (DTMC) [9]. Probabilistic model checking [11] is a verification technique for the probabilistic systems. Probabilistic Computation Tree Logic (PCTL) [12] is a probabilistic temporal logic that extends CTL to express probabilistic properties, which can be interpreted over the DTMC model. As in Filieri's study [8], which our study is based on, we assume that the system model is described as a DTMC and requirements are expressed in PCTL. Daws [13] also gives a theoretical background of parametric model checking of PCTL formulae over DTMC models. In this section, we first present the verification process that Filieri et al. proposed and then briefly explain DTMC and PCTL. The last subsection explains how Filieri et al. construct expressions at design time.

A. Efficient Run-time Verification

Figure 1 illustrates an efficient run-time verification process that Filieri et al. proposed in [8]. To realize on-the-fly analysis, the process separates a model checking activity into two steps, ones executed at design time and run-time. In the step at design time, expressions for verification are generated from a system model represented as DTMC and requirements expressed in PCTL. Transition probabilities in DTMC are represented as real values $[0, 1]$ or variables. A variable in the DTMC model represents that a transition probability is unknown at design time. In the step at run-time, values of the variables are identified by monitoring the system or its environment, and then the system evaluates the expression by substituting the values to the transition variables to verify whether the system model satisfies requirements or not.

B. Discrete Time Markov Chain

A DTMC model is a state transition diagram that has state transitions represented as probabilities. DTMC is a stochastic process [14] that has transitions from one state to another state on a state space. DTMC must respect the following property of Markov chains: the probability distribution of the next state depends only on the current state and not on the sequence of events that preceded the current state. We consider finite and time-homogeneous DTMC models, i.e., where the state space is finite and transition probabilities between states do not change with time, in this paper.

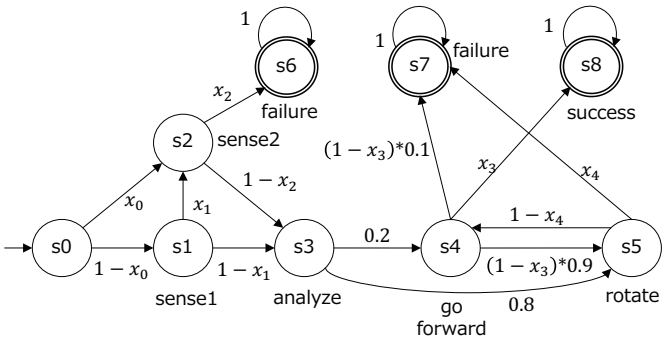

Fig. 2. An example of DTMC model: a cleaning robot.

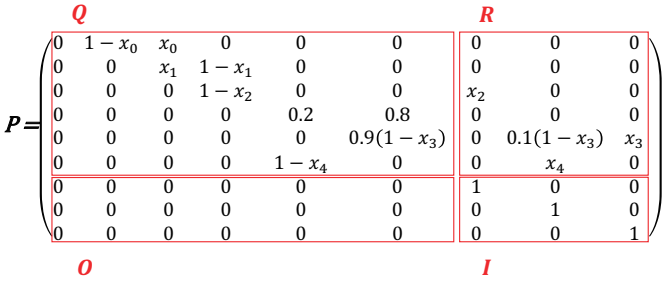

Fig. 3. The matrix P and its sub-matrices Q, R, O, and I of the DTMC model for the cleaning robot illustrated in Figure 2

Formally, a DTMC is represented as a tuple (S, S_0, P, L), where S is a set of states; S_0 ($\subseteq S$) is a set of initial states; $P : S \times S \rightarrow [0, 1]$ is a stochastic matrix, in which an element of $P(s_i, s_j)$ represents the probability that the next state of the process will be s_j given the current state is s_i; $L : S \rightarrow 2^{AP}$ is a labeling function, where AP is a set of atomic propositions that represent basic properties. A state $s \in S$ with $P(s, s) = 1$ is called an *absorbing state*. A state that is not an absorbing state is called a *transient state*. A DTMC is said to be absorbing if it contains at least one absorbing state and from every state it is possible to go to an absorbing state. In order to apply the Filieri's approach, the DTMC must be absorbing. It should be noted that the approach can sometimes be applied to a non-absorbing DTMC with some modifications to the model. For example, if the DTMC contains an absorbing connected component that has no absorbing state, then replacing the component with an absorbing state yields an alternative DTMC model that is absorbing.

We show an example of a DTMC model in Figure 2. This model represents a behavioral model for a cleaning robot. Since some of the transition probabilities are determined by monitoring the environment at run-time and cannot be determined at design time, these transitions are described as variables. For example, the cleaning robot detects obstacles by using sensors that the robot has and decides a direction to move by using data collected by sensors. The robot moves in different directions by executing going forward and turning commands. Since which sensor is used is adjusted at run-time, transition probabilities to sensing states are represented as variables x_0, ..., x_3.

A DTMC model can be represented as a matrix. For example, the matrix illustrated in Figure 3 represents the

DTMC model of our cleaning robot example (Figure 2). An absorbing DTMC that has r absorbing states and t transient states is represented by the matrix P that is in the following canonical form:

$$P = \begin{pmatrix} Q & R \\ O & I \end{pmatrix} \tag{1}$$

where I is an r by r identity matrix, O is an r by t zero matrix, R is a t by r nonzero matrix and Q is a t by t matrix. The element q_{ij} of the matrix Q is the transition probability from the transient state s_i to the transient state s_j. The element r_{ik} of the matrix R is the transition probability from the transient state s_i to the absorbing state s_k. An absorbing state has probability 1 from the absorbing state to itself. Therefore, the probabilities of absorbing states to themselves are represented as the identity matrix I. Since transition probability from an absorbing state to a transient state is 0, O is a zero matrix.

The transition probability from the transient state s_i to the transient state s_j in two steps is calculated by $\sum_{s_x \in S} P(s_i, s_x) \cdot P(s_x, s_j) = \sum_{s_x \in S} Q(s_i, s_x) \cdot Q(s_x, s_j)$. The k-step transition probability, with which the state s_j is reached from the state s_i in k steps, is the element q_{ij}^k of the matrix Q^k. Since the element q_{ij}^0 (of the matrix Q^0) represents the transition probability from the state s_i to the state s_j in zero steps, this matrix Q^0 is a t by t identity matrix. Due to the fact that R must be a nonzero matrix, Q has uniform-norm strictly less than 1, thus $Q^n \to 0$ when $n \to \infty$, which implies that eventually the process will be absorbed with probability 1.

C. Probabilistic Computational Tree Logic

The requirements are expressed in probabilistic computational tree logic (PCTL), which can be used to describe the properties of the system. Filieri et al. proposed an efficient approach to verify whether DTMC models satisfy requirements expressed by PCTL. PCTL is an extension of Computational Tree Logic (CTL) that allows probabilistic quantification of described properties. The PCTL formula [11] is defined by the following Φ:

$$\Phi \quad ::= \quad true \mid a \mid \Phi \ \wedge \ \Phi \mid \neg \Phi \mid \mathcal{P}_{\bowtie p}(\varphi) \tag{2}$$

$$\varphi \quad ::= \quad X \ \Phi \mid \Phi \ U \ \Phi \mid \Phi \ U^{\leq t} \ \Phi \tag{3}$$

where a is an atomic proposition. Probabilistic operator $\mathcal{P}_{\bowtie p}(\varphi)$ represents whether the expression φ satisfies the constraint $\bowtie p$ or not, where $p \in [0, 1]$, $\bowtie \in \{\leq, <, \geq, >\}$. Operators X and U, which appear in path formulae φ, represent "next" and "until", respectively. For example, $P_{<0.01}(X \ s = 1)$ means that the reachability probability from the current state to the next state s_1 is less than 0.01.

In Section IV, we conduct an experiment to evaluate our approach by observing the computational time for the determination of reachability property. The reachability property states that a target state is eventually reached from a given initial state. Many safety properties can be checked by reachability property verifications. In most cases, the state to be reached is one of absorbing states. An example of a PCTL formula that represents a reachability property is as follows:

$$P_{\leq 0.01}(true \ U \ s = 6) \ (= P_{\leq 0.01}(F \ s = 6)) \tag{4}$$

Expression (4) states that the probability from an initial state to the absorbing state s_6 has to be less than 0.01. Since many practical requirements can be specified as the reachability of an absorbing state [15], we focus on the reachability properties in this paper.

D. Expression Generation

As we described, the probability of moving from the transient state s_i to the absorbing state s_k in one step is represented as the element p_{ik} of the matrix P. In the case of more than 1 step, we have to compute the probability of the transition from a transient state to another transient state and the transition from the transient state to the destination state. The transition probability in two steps is the element q_{ij}^2 of the matrix $Q^2 (= Q \times Q)$, and the transition probability in k steps is the element q_{ij}^k (of the matrix Q^k). Therefore, the expected number of visits to the transient state s_j before arriving at an absorbing state, starting from the state s_i is the (i, j)-th element of the matrix $N = I + Q + Q^2 + Q^3 + \cdots = \sum_{k=0}^{\infty} Q^k$. The probability from the transient state s_i to the absorbing state s_k in some steps is defined as the element b_{ik} of the matrix B:

$$B = N \times R \tag{5}$$

The computation for the matrix B requires the computation for the matrix N. We can replace N as follows:

$$\begin{aligned} N &= I + Q + Q^2 + Q^3 + \cdots \\ &= \sum_{k=0}^{\infty} Q^k \\ &= (I - Q)^{-1} \end{aligned} \tag{6}$$

Since the matrix N is the inverse of $(I - Q)$, we calculate the element n_{ij} of the matrix N as follows:

$$n_{ij} = \frac{1}{det(W)} \cdot \alpha_{ji}(W) \tag{7}$$

where W is $I - Q$, and α_{ij} is the cofactor. Using Expression (5) and (7), the element b_{ij} of the matrix is calculated as follows:

$$\begin{aligned} b_{ik} &= \sum_{x \in 0 \ldots t-1} n_{ix} \cdot r_{xj} \\ &= \frac{1}{det(W)} \sum_{x \in 0 \ldots t-1} \alpha_{xi}(W) \cdot r_{xj} \end{aligned} \tag{8}$$

Here, the computation of b_{ik} requires the computation of determinants. Determinants are calculated by applying Laplace expansion [16] and LU-decomposition [17] (Figure 4). Laplace expansion removes variables from a matrix; LU-decomposition, on the other hand, computes determinants of the matrix that contains no variables. The determinant $|A|$ of the $n \times n$ matrix A is the sum of the product of the element a_{ij} by each determinant of n sub-matrices of A with size $(n-1) \times (n-1)$. The expression of $|A|$ with the i-th row expanded is as follows:

$$|A| (= det(A)) = \sum_{j=0}^{n} (-1)^{i+j} a_{ij} |A'|_{ij} \tag{9}$$

A'_{ij} is the $(n-1) \times (n-1)$ sub-matrix generated by removing the i-th row and the j-th column of the matrix A.

As we mentioned, the computation of reachability property requires the determinant computation. The determinant computation is executed by using LU-decomposition and Laplace

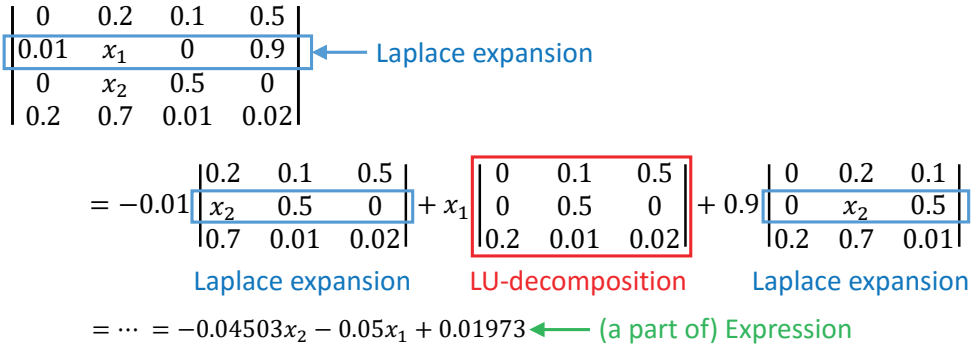

Fig. 4. Laplace expansion and LU-decomposition.

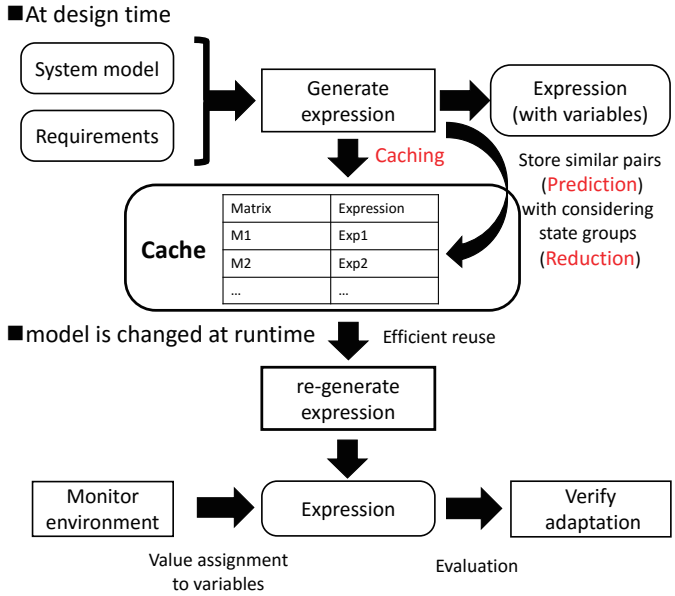

Fig. 5. Our expansion: efficient reuse of the pairs of matrix and expression with three strategies, i.e., *caching, prediction, and reduction.*

expansion; while the former requires $O(n^3)$ computational cost, the latter requires $O(n!)$, that is, considerably larger computational cost. The rows that include variables representing unknown parameters are expanded by using Laplace expansion. The high number of times of Laplace expansions leads to a huge computational cost. Models of systems that have to deal with uncertain environments, such as self-adaptive systems, usually have variables that are represented as unknown parameters in the matrices. The large number of variables leads to the large number of times of Laplace expansions and large computational time.

III. Our approach

The advantage of Filieri's approach is that it can shift the large computational cost from run-time to design time. This process provides an efficient run-time verification mechanism under the assumption that the structure of the system model, i.e., the DTMC model, is not changed. On the other hand, if the structure is changed, such as states and transitions addition, the pre-computed expressions no longer correspond to the new model.

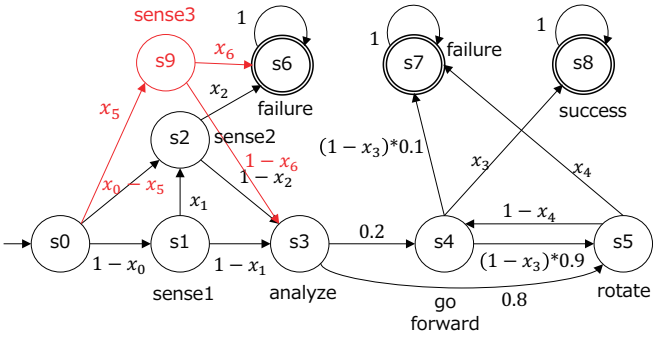

Fig. 6. The DTMC model of our cleaning robot after adaptation. A new state s_{11} for sensing another data from an environment and relevant transitions are introduced.

For example, Figure 6 shows a model after an adaptation that causes to add state and transitions to the model illustrated in Figure 2. In such a case, the system has to re-execute the pre-computation process at run-time. This re-execution usually requires large computational time. Especially, Laplace expansions in the pre-computation process require a large amount of computational time.

The goal in this paper is to reduce the number of executing Laplace expansions at run-time. Figure 5 illustrates the overview of our approach. In this paper, we propose the following three strategies:

- **Caching:** To avoid the re-execution of Laplace expansion, we store pairs of sub-matrix and the corresponding expression for re-use into the memory (cache). When the system needs to re-generate an expression, it searches for the matrix from the cache. If the system finds it, the matrix is replaced with the stored expression with no calculation.
- **Prediction:** Since a DTMC model is changed by adaptation, the cache hit ratio may be decreased when we search for sub-matrices of the new DTMC model. In order to increase hit ratio of caching, we store additional pairs that are similar to the stored pairs based on the caching strategy.
- **Reduction:** Caching and prediction strategies require an additional memory space. We need to reduce the increase of the required space. In this paper, we try to

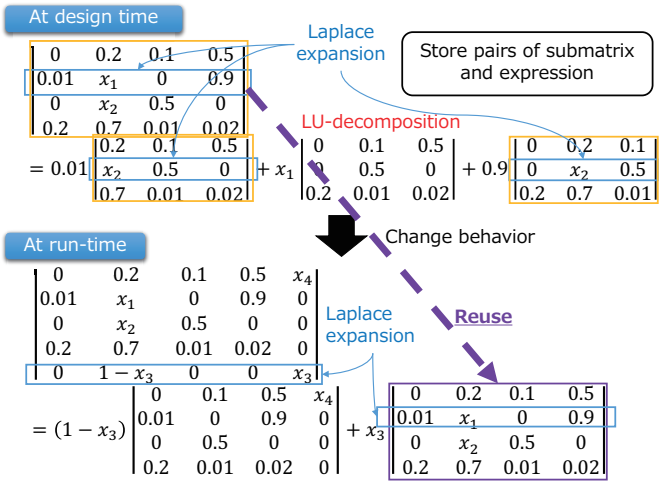

Fig. 7. Caching.

realize the reduction on the basis of *grouping*.

The following subsections briefly explain these strategies.

A. Caching

In many cases, self-adaptive systems change only a part of their behaviors. Even though the systems requires re-generation of expression for verification at run-time, the most part of the system models are not changed in many cases.

Figure 7 illustrates the caching strategy. We store the pairs of sub-matrix and the corresponding (sub-)expression at design time. Since LU-decomposition does not require the large computational cost, we store the sub-matrices that require Laplace expansion. When the system needs to re-generate expressions at run-time, it searches for the matrix to be processed from the cache. If the system finds the same matrix in the cache, the matrix is replaced with the paired sub-expression.

B. Prediction

Effectiveness of caching depends on the size of matrix to be replaced with an expression. Even though we can replace a small size of sub-matrix with the corresponding expression, the improvement of computational cost is limited. We should find as large a size of matrix as possible.

In order to match a large size of matrix, we additionally register matrices that are similar to the matrix representing the system model before adaptation.

In this paper, we focus on the behavioral changes such that new states and transitions are added. For example, the model shown in Figure 6 is an example of such cases. Figure 8 is the corresponding matrix. The prediction strategy makes such matrices at design time by adding one state and one transition from the existing matrix. These matrices can be constructed as follows:

1) Insert a new row and column in the rear of Q. Let the inserted row and column be the i-th row and the j-th column. All elements in the added row and column are 0.
2) Choose a non-zero value from the elements in Q. Let the selected element be a_{kl} and its value be α_{kl}.
3) Introduce a new variable x_{new} and put this variable at a_{kj}. Since the sum of transition probabilities in each row should be 1, we change the value α_{kl} at a_{kl} to $\alpha_{kl} - x_{new}$. We regard this matrix as a matrix similar to the existing matrix.
4) Repeat Steps 2 and 3 and generate matrices by changing a non-zero element in Q at Step 2 until all of the non-zero elements in Q are selected.

For example, our prediction strategy can make the matrix illustrated in Figure 8 from the matrix illustrated in Figure 3, that is, the model before adaptation.

C. Reduction

Caching and prediction strategies are expected to require a large amount of memory space. To restrain the increase of memory usage, we try to reduce the candidate matrices on the basis of grouping. The reduction strategy defines sets of states in DTMC model as groups. This strategy removes the matrices that the prediction strategy generated by injecting transitions over different groups. This restriction saves certain degree of matrix generation.

Considering the cleaning robot example illustrated in Figure 2, we can define groups for *sensing* (the set of states s_0 to s_3 and the state s_6) and *migration* (the set of states s_4, s_5, s_7, and s_8). When we prepare the similar matrix added a new state into the sensing group, we consider new transitions from the states within the sensing group, i.e., s_0, s_1, s_2, s_3, and s_6, to the new state and transitions from the new state to these states. The reduction strategy prevents generating similar matrices that have less possibility of re-use.

IV. Evaluation

In order to evaluate the effectiveness of our strategies, we conducted preliminary experiments on randomly generated DTMC models. We implemented three strategies in Java and compared the computational time and memory size of four methods of re-generation as follows:

- Method 1: baseline (no caching), denoted as `none` in Figures 9 to 12;
- Method 2: caching, denoted as `C` in the figures;
- Method 3: caching and prediction, denoted as `CP` in the figures;
- Method 4: caching, prediction, and reduction, denoted as `CPR` in the figures.

As problem instances, we randomly generated DTMC models with the size ranging from 20 to 40 states increased by five, including two absorbing states. Some generated models contain loops. In these DTMC models, the number of outgoing transitions of each state follows a Gaussian distribution with mean 10 and standard deviation 2. Initial DTMC models, that is, the system models before adaptation, contain one or two variables as transition probabilities, and adaptation causes to introduce two variables into the models. We divided all states

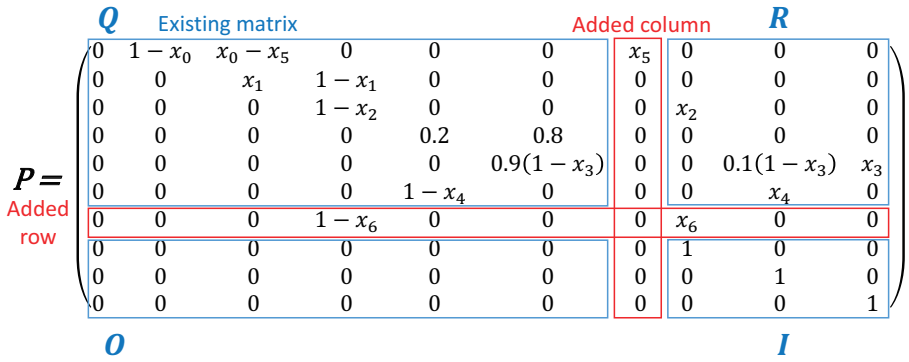

Fig. 8. The matrix of the DTMC model for the cleaning robot after adaptation. Our prediction strategy makes such a matrix from the existing matrix.

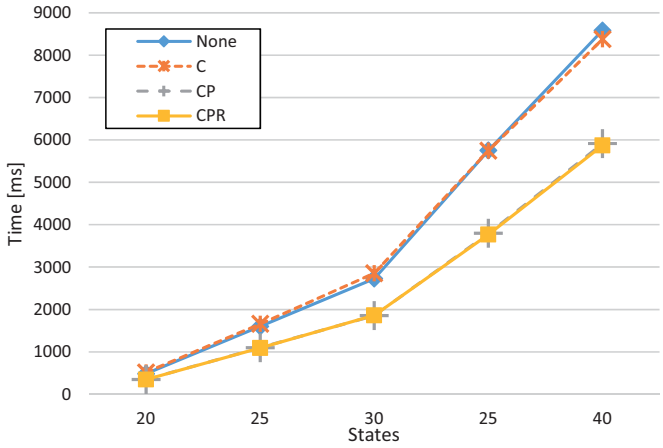

Fig. 9. Computational time when the model before adaptation has one variable and the model after adaptation has three variables. X-axis represents the number of states in the DTMC model, while Y-axis represents computational time [ms].

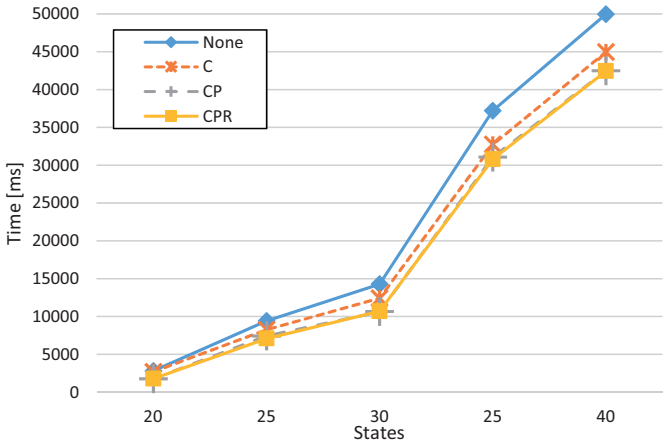

Fig. 10. Computational time when the model before adaptation has two variables and the model after adaptation has four variables.

into groups of size 10. We compared the averages of 10 runs for each configuration.

Figures 9 and 10 show the computational time to re-generate an expression. We can find that caching (Method 2) speeds up expression re-generation (faster than Method 1) but

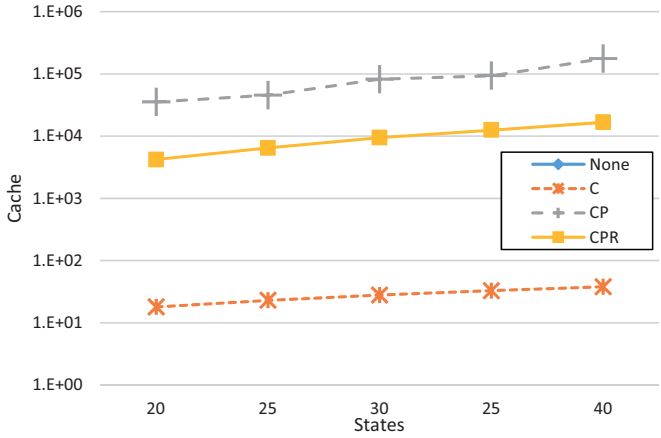

Fig. 11. Cache size when the model before adaptation has one variable and the model after adaptation has three variables. X-axis represents the number of states in the DTMC model, while Y-axis represents the number of the stored expressions in logarithmic scale. Since Method 1, represented as `None` in legend, does not use cache, it does not appear in this graph. (Cache size is zero.)

sometimes it does not affect computational time. Prediction (Method 3) and reduction (Method 4) further make the computation faster.

As for the cache size illustrated in Figures 11 and 12, on the other hand, prediction (Method 3) requires the largest memory size. Reduction (Method 4) partly reduces the memory size for caching.

The results of the experiments shows that the *prediction* strategy, on the basis of *caching*, is effective to reduce the computational time and the *reduction* strategy works to prevent the increase of memory size for caching. These strategies partially work; however, we also find that the effects of these strategies are still limited.

The current design of prediction with reduction prepares similar matrices for all elements in the same group in Q. We could make more precise prediction using the semantic of the system model. As for the memory size, our current reduction strategy is effective when behavioral changes are limited within the group. Since the effectiveness of reduction strategy is different between the cases illustrated in Figures 11 and 12, we should improve the applicability of reduction strategy. For further memory usage reduction, we could discard

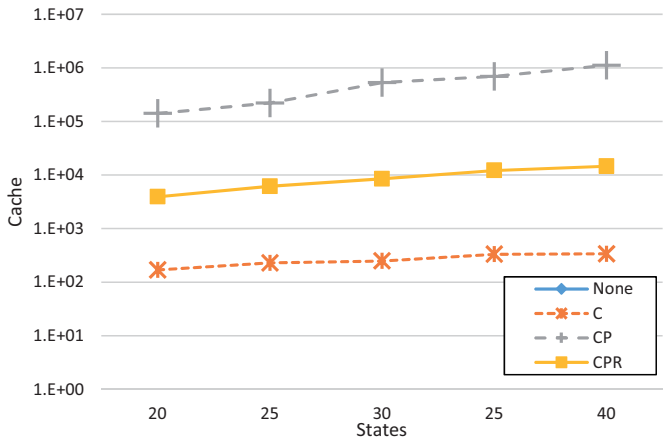

Fig. 12. Cache size when the model before adaptation has two variables and the model after adaptation has four variables. Since Method 1, represented as `None` in legend, does not use cache, it does not appear in this graph. (Cache size is zero.)

small matrices that do not substantially affect computational time and redundant ones.

When using the proposed strategies, software developers should estimate the benefits of the strategies because they are still limited. The time required to generate expressions is reduced in the range of 10 to 30%. Also memory consumption should be taken into account. Even though prediction and reduction strategies are effective for reducing computational time, they consume a large amount of memory; this trade off should be carefully considered.

V. Related Work

Runtime verification, such as one in [7], plays a key role in dynamically adaptive software systems. Some work, in particular, focuses on the efficiency of runtime verification. Gerasimou et al. [18] evaluated the effectiveness of caching, look-ahead, and nearly-optimal techniques on the continuous-time Markov chain (CTMC) model. While this technique is designed on the assumption that verifications are continuously executed at run-time with using previous results, our approach, on the other hand, is designed on the assumption that computational results to be re-used have been prepared at design time.

Kwiatkowska et al. [19] proposed an incremental quantitative verification of probabilistic models. This method uses the previous verification results that are acquired at run-time to speed up run-time verification. This approach can also deal with the structural changes of models. While this approach requires the analysis of change impact on the model before re-use of the previous verification results, our approach determines where the re-use can be applied by only checking cache hit, which can be implemented by a simpler mechanism.

VI. Conclusion and Direction for future work

We presented the current state of our approach that supports run-time verification when the structure of the system model is changed by adaptation. In order to reduce the computational time for re-generating expressions, we proposed three strategies, i.e., *caching*, *prediction*, and *reduction*. The preliminary evaluation demonstrated the possibilities of our strategies.

From the experimental results, we identified the directions for future work. First, we should address the reduction of cache size. Some Markov model simplification techniques, such as ones based on SCC (strongly connected components) reduction [20], partial order reduction [21], and bisimulation [22], can be beneficial to reduce cache size; however, excessive reduction could prevent our strategies from reusing stored matrices for regenerating expressions.

Second, we plan to improve the precision strategy of the prediction strategy. The improvement of prediction helps to keep the cache size from increasing. We also plan to improve the applicability of reduction strategy. One direction is discarding small size matrices and redundant ones. Third, after the sufficient improvement of our strategies, we will take into account different adaptation patterns, such as state removal or larger changes in the system model. In order to evaluate our approach on real world examples, we plan to embed the verification mechanism in our programming framework [23] for self-adaptive systems.

Acknowledgment

This work was supported by JSPS Grants-in-Aid for Scientific Research (No. 15K00097). The Telecommunications Advancement Foundation also supported this work.

References

[1] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, and et al., "Software engineering for self-adaptive systems: A research road map," in *Dagstuhl Seminar Proceedings 08031*, 2008, pp. 1 – 13.

[2] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, and et al., "Software engineering for self-adaptive systems: A second research roadmap (draft version of may 20, 2011)," in *Dagstuhl Seminar 10431*, 2011.

[3] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the 28th international conference on Software engineering (ICSE '06)*. ACM, 2006, pp. 371–380.

[4] G. Blair, N. Bencomo, and R. B. France, "Models@ Run.Time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.326

[5] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Communications of the ACM*, vol. 55, no. 9, pp. 69–77, Sep. 2012. [Online]. Available: http://doi.acm.org/10.1145/2330667.2330686

[6] 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.

[7] M. U. Iftikhar and D. Weyns, "ActivFORMS: Active Formal Models for Self-adaptation," in *Proc. of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, 2014, pp. 125–134. [Online]. Available: http://doi.acm.org/10.1145/2593929.2593944

[8] A. 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.

[9] K. Goševa-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Perform. Evaluation*, vol. 45, no. 2-3, pp. 179–204, Jul. 2001. [Online]. Available: http://dx.doi.org/10.1016/S0166-5316(01)00034-7

[10] K. Ogawa, H. Nakagawa, and T. Tsuchiya, "An experimental evaluation on runtime verification of self-adaptive systems in the presence of uncertain transition probabilities," in *Proc. of Software Engineering and Formal Methods - SEFM 2015 Workshops*, ser. LNCS, vol. 9509, 2015, pp. 253–265.

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

[12] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994. [Online]. Available: http://dx.doi.org/10.1007/BF01211866

[13] C. Daws, "Symbolic and parametric model checking of discrete-time markov chains," in *Proc. of the First International Conference on Theoretical Aspects of Computing (ICTAC'04)*, ser. ICTAC'04. Springer-Verlag, 2005, pp. 280–294. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31862-0_21

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

[15] L. Grunske, "Specification patterns for probabilistic quality properties," in *Proc. of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, 2008, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368094

[16] H. Rose and H. Rose, *Linear Algebra: A Pure Mathematical Approach*. Springer, 2002. [Online]. Available: https://books.google.co.jp/books?id=mTdAj-Yn4L4C

[17] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[18] S. Gerasimou, R. Calinescu, and A. Banks, "Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration," in *Proc. of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, 2014, pp. 115–124. [Online]. Available: http://doi.acm.org/10.1145/2593929.2593932

[19] M. Kwiatkowska, D. Parker, and H. Qu, "Incremental quantitative verification for markov decision processes," in *Proc. of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, ser. DSN '11. IEEE Computer Society, 2011, pp. 359–370. [Online]. Available: http://dx.doi.org/10.1109/DSN.2011.5958249

[20] F. Ciesinski, C. Baier, M. Grosser, and J. Klein, "Reduction techniques for model checking markov decision processes," in *Proc. of the Fifth International Conference on Quantitative Evaluation of Systems (QEST '08)*, 2008, pp. 45–54.

[21] H. Hansen, M. Kwiatkowska, and H. Qu, "Partial order reduction for model checking markov decision processes under unconditional fairness," in *Proc. of the 2011 Eighth International Conference on Quantitative Evaluation of SysTems (QEST '11)*. IEEE Computer Society, 2011, pp. 203–212. [Online]. Available: http://dx.doi.org/10.1109/QEST.2011.35

[22] J.-P. Katoen, T. Kemna, I. Zapreev, and D. N. Jansen, "Bisimulation minimisation mostly speeds up probabilistic model checking," in *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, ser. TACAS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 87–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=1763507.1763519

[23] H. Nakagawa, A. Ohsuga, and S. Honiden, "Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops," in *Proc. of IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems (SASO'12)*. IEEE, sept. 2012, pp. 59 – 68.