Article

Runtime Verification in Uncertain Environment Based on Probabilistic Model Learning

Ge Zhou [1,†,‡], Wanwei Liu [1,‡] and Wei Dong [1,*]

[1] Department of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China; zhouge12@nudt.edu.cn

* Correspondence:wdong@nudt.edu.cn;

† Current address: Yanwachi Main Street 109, Changsha 410073, Hunan, China.

‡ These authors contributed equally to this work.

Version April 10, 2019 submitted to Preprints

Abstract: Runtime verification (RV) is a lightweight approach to detecting temporal errors of system at runtime. It confines the verification on observed trajectory which avoids state explosion problem. To predict the future violation, some work proposed the predictive RV which uses the information from models or static analysis. But for software whose models and codes cannot be obtained, or systems running under uncertain environment, these predictive methods cannot take effect. Meanwhile, RV in general takes multi-valued logic as the specification languages, for example the *true*, *false* and *inconclusive* in three-valued semantics. They cannot give accurate quantitative description of correctness when *inconclusive* is encountered. We in this paper present a RV method which learns probabilistic model of system and environment from history traces and then generates probabilistic runtime monitor to quantitatively predict the satisfaction of temporal property at each runtime state. In this approach, Hidden Markov Model (HMM) is firstly learned and then transformed to Discrete Time Markov Chain (DTMC). To construct incremental monitor, the monitored LTL property is translated into Deterministic Rabin Automaton (DRA). The final probabilistic monitor is obtained by generating the product of DTMC and DRA, and computing the probabilities for each state. With such method, one can give early warning once the probability of correctness is lower than pre-defined threshold, and have the chance to do adjustment in advance. The method has been implemented and experimented on real UAS (Unmanned Aerial Vehicle) simulation platform.

Keywords: Runtime Verification, Probabilistic Monitor, Markov Chain, ω-automata

1. Introduction

1.1. Motivation and Contribution

Runtime verification (RV) is a lightweight formal verification technique in software verification [1]. It decides whether the system's running satisfies specific properties only based on the current observed trace [2]. RV is closely related to model checking and testing, which are two commonly used methods to evaluate the credibility of software. However, when the software to be verified is very complicated, model checking will encounter the problem of state explosion [3], and testing will also be difficult to cover most paths of the software [4]. RV complements model checking and testing because it makes conclusions of the properties using only observed trace, which is insensitive to software scale. RV can also monitor the properties related to operating environment after software is deployed on actual platform.

Traditional RV techniques only give judgement with observed trace, while predictive RV [5] will predict the running trend of the target system in advance. This feature can provide great potential to avoid software failures rather than only find them. One main existing predictive RV method uses *predictive words* [6] to realize advance prediction. The so-called predictive words are auxiliary monitor information generated from the models or codes of target software with the methods such as model abstraction or static analysis. However, for some historical legacy software and black box systems, or the systems running in uncertain environment, these methods are not applicable.

RV in general takes multi-valued logic as the specification language, and the monitoring process stops whenever the correctness values (i.e., *true* or *false*) are encountered. A major drawback of these methods is that it can not provide accurate quantitative description for the satisfaction of property. To resolve this problem and make RV suitable for infinite trace semantics, temporal logic LTL_3 adds *inconclusive* to truth values and corresponding monitor generation method was proposed [7]. But its evaluation value of a formula remains to be *unknown* when *inconclusive* is encountered, which may be the most situations in monitoring process. Hence, in these settings, one cannot give a precise predication of the trend of correctness. For such situation, if a probabilistic value can be computed for judgement *inconclusive* to evaluate the satisfaction of the properties in a quantitative way, it will compensate the deficiency of the existing predictive RV methods. Furthermore, for different target systems and different properties, we can set corresponding thresholds for acceptance of probabilistic values based on requirement or expert experience. When the system with a probabilistic monitor detects that the value of current trace exceeds the threshold, alarm or control operations can be issued to steer the running of system.

In this paper, we propose the method of learning probabilistic model from historical traces which include system and environment events, and generating the monitor which can give the probability that the property will be satisfied (or violated) when new state of current trace is observed. For this purpose, a Hidden Markov Model (HMM) is learned from historical traces and translated to Discrete Time Markov Chain (DTMC), and the Deterministic Rabin Automaton (DRA) is generated from the property in Linear Temporal Logic (LTL). Then the probabilistic monitor will be generated using DTMC and DRA. Such probabilistic RV method can predict the trend of the target system by quantitatively judging the extent to which the current system state satisfies the monitored property. Compared with previous predictive RV methods, in our approach the model or code of target system is not needed and the environment is also be considered. The probabilistic monitor can also give the guidance information for software execution. When the software deviates from the expected properties, the running of target system can be adjusted by predefined behavior to avoid malfunction. The corresponding tool of learning and generating probabilistic monitor is implemented and the experiments show the effectiveness of the proposed method.

The paper is organized as follows. Section 1 introduces the motivation and related work, and section 2 presents some basic concepts. Section 3 gives the method of learning HMM and DTMC, while section 4 elaborates the way to generate probabilistic monitor. Section 5 describe the implements and experiment. The paper is concluded in section 6.

1.2. Related Work

Verifying probabilistic properties is firstly considered in probabilistic model checking, which has been studied for a long time with adequate theoretical results and important application fields. For example, in [8], probabilistic model checking is applied in provisioning of cloud resources, which shows good results in experiments. The work in [9] discuss two categories of probabilistic model checking when applied in self-adapted systems. Learning is used in [10] to get the abstract model of stochastic systems without source code for statistical model checking.

In recent years, combining RV with probability and statistics has become a novel research direction. To do statistical checking of probabilistic properties at runtime, [11] provides a method to decompose a trace into several samples based on specification of two kinds of events. In [12], the probabilistic model is extracted, and traditional instrumentation and monitoring methods in RV are combined to evaluate the property probabilities in quantitative and qualitative ways. [13] did similar work upon the framework of WMI and .NET. Above works mainly study the methods of determining if a probabilistic property will be satisfied for observed trace, but cannot determine the probability that a deterministic property will be satisfied or violated in uncertain environment.

Traditional RV methods cannot predict whether the future running of the target system satisfies given properties. Predictive RV tries to extend this ability, such as Anticipatory Active Monitoring [14] proposed by us. The main idea is to process the models or codes of the software through static analysis or other methods, so as to obtain necessary information to assist runtime monitor to make decision as early as possible. One of our previous work in predictive RV is based on system model, as the framework shown in Figure 1A. $\mathcal{G}$ is the model of software to be monitored. During runtime, the monitor $\mathcal{M}$ generated from LTL property will use the information of model $\mathcal{G}$ to predict whether the current system trace $\bar{\rho}$ will violate the property in the future. When there exist violation paths from current state, the active monitor will generate a corresponding intervention behavior $V(\bar{\rho})$ which can guide the running to correct paths and feed it back to the running system $\mathcal{P}$.

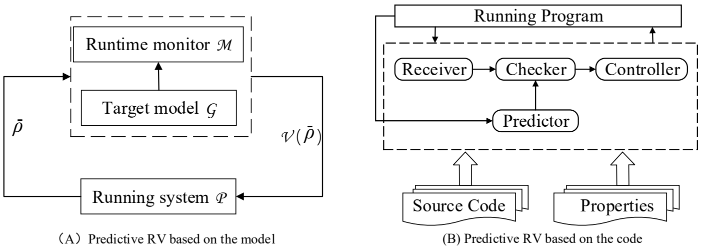

Runtime monitor $\mathcal{M}$
Target model $\mathcal{G}$
$\bar{\rho}$
$\mathcal{V}(\bar{\rho})$
Running system $\mathcal{P}$
（A）Predictive RV based on the model
Running Program
Receiver
Checker
Controller
Predictor
Source Code
Properties
(B) Predictive RV based on the code

Figure 1. Two types of existing predictive RV methods.

For software without models, we also proposed another predictive RV method which depends on code static analysis [6], as shown in Figure 1B. Control Flow Graph (CFG) of the code is extracted before software is deployed. For each control node in CFG, the event sequences related to the property in its scope are generated, and the variables whose values will be calculated at runtime and used in branch decision expression are also recorded. At runtime, the monitor will predict the property violation based on these event sequences and variable values at current state. Although the method can predict the violation more accurately since branch decision is determined at runtime, the cost may be huge for complex software.

One obvious shortcoming of above predictive RV methods is that they are not suitable for software without models or codes, or systems running in uncertain environment. These problems will be focused on in this paper.

2. Preliminaries

2.1. Linear-time Temporal Logic

Linear-time Temporal Logic (LTL) is usually used to specify the temporal behavior and properties of system. Assuming that $\mathcal{P}$ is a set of atomic propositions, and $\Sigma = 2^{\mathcal{P}}$ denotes the finite alphabet, LTL formulae can be defined as

$$\varphi ::= \; p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2$$

where $p \in \mathcal{P}$. There are some standard derived operators, such as:

$$\top \equiv \; p \vee \neg p$$
$$\varphi_1 \wedge \varphi_2 \; \equiv \; \neg(\neg\varphi_1 \vee \neg\varphi_2)$$
$$\varphi_1 \rightarrow \varphi_2 \; \equiv \; \neg\varphi_1 \vee \varphi_2$$
$$F\varphi \; \equiv \; \top U\varphi$$
$$G\varphi \; \equiv \; \neg(F\neg\varphi)$$
$$\varphi_1 R \varphi_2 \; \equiv \; \neg(\neg\varphi_1 U \neg\varphi_2)$$

Semantics of LTL formulae are defined with an infinite trace $\pi \in \Sigma^\omega$ and a position i. The i-th letter $\pi(i)$ of the trace π is a subset of $\mathcal{P}$, which can be viewed as an assignment to $\mathcal{P}$. The standard semantics of the LTL formulae are defined as follows [7]:

$$\begin{array}{lcll}
\pi, i & \models & p \in \mathcal{P} & \Leftrightarrow \ p \in \pi(i) \\
\pi, i & \models & \neg\varphi & \Leftrightarrow \ \pi, i \not\models \varphi \\
\pi, i & \models & \varphi_1 \vee \varphi_2 & \Leftrightarrow \ \pi, i \models \varphi_1 \ or \ \pi, i \models \varphi_2 \\
\pi, i & \models & X\varphi & \Leftrightarrow \ \pi, i+1 \models \varphi \\
\pi, i & \models & \varphi_1 U \varphi_2 & \Leftrightarrow \ \exists j \geq i : \pi, j \models \varphi_2 \ and \\
 & & & \quad \forall i \leqslant k < j : \pi, k \models \varphi_1
\end{array}$$

2.2. RV and Monitoring Semantics

RV is a lightweight formal verification method that only focuses on whether the current execution of system meets certain properties. At runtime, a system trace can be treat as a finite sequence composed of system states, and different definitions of state should be given for different concerns. At hardware level, the state of the system should be defined with the combination of values of registers, memories and so on. At software level, the system state is usually defined by the locations of the program, the values of variables, the events of function calling and so on.

The current system execution at runtime is a finite prefix of the infinite trajectory. The goal of RV is to check whether this prefix satisfies the given property. Commonly a monitor will be generated from the given property according to a specific method to do this checking. Thus, from perspective of the formal language, RV solves the problem whether a given word is included in the language corresponding to the given property. Therefore, the monitor is the device that reads a finite trace and yields a certain verdict.

Monitoring semantics [15] is a set of advanced logical protocols that formally define what decision the monitor makes in different situations. Different monitoring semantics may get different conclusions. The standard semantics of LTL is defined on infinite traces, which can not be applied directly to finite traces. For two-valued semantics of LTL, there are the following three methods to solve this problem [16]. Given a finite trace μ and a LTL formula φ,

(1) Weak semantics: φ is violated iff μ is a bad-prefix of it. Namely, for any infinite word ν, we have $\mu\nu \not\models \varphi$.

(2) Strong semantics: In this case, φ is satisfied iff μ is a good-prefix of it. That is, $\mu\nu \models \varphi$ for every infinite word ν.

(3) Multi-value semantics: There exist consistency problems in two-valued semantics of LTL. Thus multi-value semantics was proposed and defined on infinite trace, such as LTL_3:

$$[\mu \models \varphi]_{LTL_3} \quad = \quad \begin{cases} true, \quad if \ \forall\sigma \in \Sigma^\omega : \mu\sigma \models_{LTL} \varphi; \\ false, \quad if \ \forall\sigma \in \Sigma^\omega : \mu\sigma \not\models_{LTL} \varphi; \\ ?, \quad otherwise. \end{cases}$$

2.3. Discrete Time Markov Chain

Discrete Time Markov Chain is a stochastic process in mathematics for discrete events. In the process, given the current state with knowledge or information, the historical state is not considered when predict the future state. It is called markov-process. In addition, it is found that no matter what state it is, the markov-process will gradually become stable after a period of time, and the state of stability is not related to the initial state.

With a countable set $\mathcal{P}$ of propositions, a discrete time markov chain (DTMC) M is a tuple (S, I, T, L), where

- S is a finite set of states;
- I is an initial distribution over S, and $\sum_{s \in S} I(s) = 1$;
- $T : S \times S \to [0, 1]$ is the transition matrix fulfilling $\sum_{s' \in S} T(s, s') = 1$ for each $s \in S$;
- $L : S \to 2^{\mathcal{P}}$ is the labelling function.

3. Learning based Probabilistic Modelling

Event is usually used to encapsulate the behavior and interactions for many software systems, and event-triggered RV is used to ensure the reliability of a system by observing the occurrence of events. During the process of monitoring the target system running in specific environment, there is a kind of events that seem to be independent of each other. However, there often exist some hidden correlations among these events.

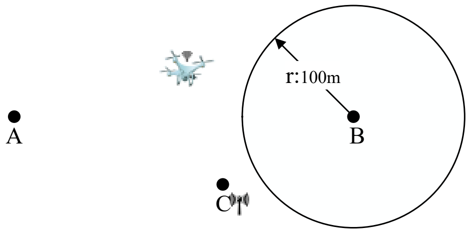


Figure 2. An example with uncertain environment in practice.

For example, as shown in the Figure 2, a drone often flies from point A to point B, and there is a control tower at point C that sends instruction to this drone. Event p is defined as *the drone reaches the area within 100 meters around point B*. The operator obtains the position information from the drone and sends instructions through the control tower. In a real environment, the communication between drone and control tower may be hindered by some obstacles, thereby leading to some flight deviation. Another event q is defined as *the drone sends the position information back to control tower C*. We have to monitor whether the mission can be completed successfully. On the surface, p and q are independent events, but these events contain potential probabilistic relationships since the landform such as mountains or interference sources nearby may affect the sending and receiving of information. This case will also be used in the following when describing the proposed method.

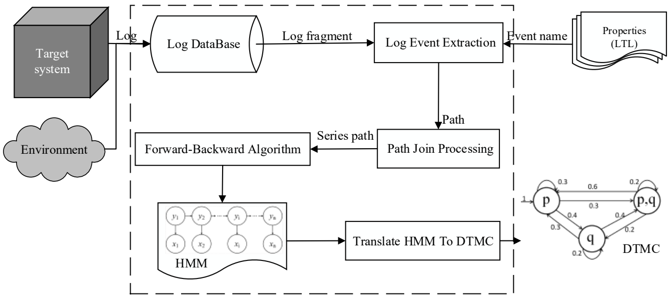


Figure 3. Framework of learning the model of target system and environment.

Figure 3 show the framework of learning the DTMC model of target system and environment. First, event traces related to the monitoring property need to be extracted from the log history. Then these paths are joint to form one trace from which the HMM will be learned according to the distribution of their occurrence. Here the forward-backward algorithm in machine learning is adopted. Finally HMM is transformed to DTMC, which will be used in monitor generation.

3.1. Problem Description

For the software without providing models and codes, running trajectory can be obtained from the historical running data. In the case of Figure 2 for example, we can randomly extract a great of trajectories from the log database to learn the model with the focus on events p and q. The traces can be expressed as the form such as: $\pi = S_{init} \rightarrow p \rightarrow p \rightarrow q \rightarrow p \rightarrow \emptyset \rightarrow \emptyset \rightarrow S_{End}$. The symbol $\emptyset$ indicates that neither p nor q occur. p or q separately shows that corresponding event occur. Symbol pq is used to represent the set $\{p, q\}$, which means the occurrence of both events p and q.

Before formalizing the problem, some definitions need to be introduced. A finite trace π is a string in Σ^* and *length* of the trace is denoted by $len(\pi)$. $\pi(i)$ denote the *i-th* location of π for each $i < len(\pi)$; therefore, $\pi(i) \subseteq \mathcal{P}$. A DTMC is a stochastic process that satisfies the following provisos:

- Proviso_1: The probability distribution of the system state at time $t+1$ is only related to the state at time t and is independent of the states before t;
- Proviso_2: The state transition from time t to $t+1$ is independent of the value of t.

Given a DTMC $M = (S, I, T, L)$ and a trace π, and $[n]$ to denote the set $\{0, \ldots, n-1\}$ for $n \in \mathbb{N}$, a mapping from π to M is defined as a function δ with type $[len(\pi)] \rightarrow S$ such that $L(\delta(i)) = \pi(i)$. Then, the probability of π w.r.t.M under δ is defined as

$$prob_{\delta}(M, \pi) = I(\delta(0)) \cdot \prod_i T(\delta(i), \delta(i+1))$$

We denote by $\Delta_{M,\pi}$ the set that comprises all mappings from π to M. We now calculate the value $sup_{\delta \in \Delta_{M,\pi}} prob_{\delta}(M, \pi)$ and the corresponding mapping. In other words, our goal is to find a mapping δ that makes $prob_{\delta}(M, \pi)$ as large as possible.

We can canonically lift this problem when we are given a (finite) path set Π. In this setting, our goal is to determine a mapping $\delta_{\pi} \in \Delta_{M,\pi}$ for each $\pi \in \Pi$ and to maximize the value of $\Sigma_{\pi \in \Pi} prob_{\delta_{\pi}}(M, \pi)$. There have been some methods can be used to solve this problem. In this paper, we propose a method by learning a (HMM) model from traces to solve this problem.

3.2. Hidden Markov Model

HMM is a directed dynamic Bayesian network diagram [17] with a simple structure that is mainly used for timing data modeling, speech recognition, and natural language processing. As shown in Figure 4, HMM has two groups of variables. The first group is called the hidden or state variables Y=$\{y_1, ..., y_n\}$, where y_i represents the hidden, unobservable state at time $t = i$. Assuming that the number of all states in the model is N, that is, the state space $\mathcal{Y}$ = $\{s_1, ..., s_N\}$, we have $y_i \in \mathcal{Y}$. The second group is called the observation variables or display states X=$\{x_1, ..., x_n\}$, where x_j represents the state that can be observed directly at $t = j$. The observed variables can either be continuous or discrete. Here we only consider the discrete variables since the systems monitored in our work are supposed discrete time systems. Assuming the number of observed variables is m, that is, the observation space $\mathcal{X}$ = $\{o_1, ..., o_m\}$, we have $x_j \in \mathcal{X}$.

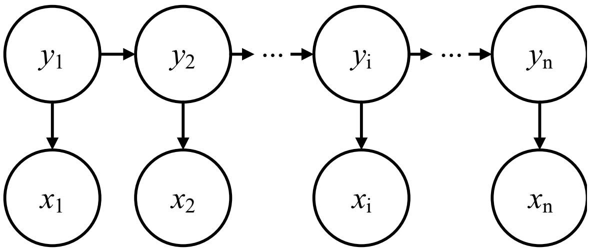


Figure 4. Mapping structure of HMM.

In Figure 4, an arrow represents the dependency among variables. HMM has the following properties:

- The change of hidden states is a DTMC, that is, process $(y_1 \rightarrow y_2 \rightarrow ... \rightarrow y_n)$ is consistent with the Markovian nature.

- The observed variable x_t is determined by the hidden variable y_t and is independent of other observed or hidden variables.

Based on above independence relationship, the joint distribution of X and Y is

$$P(x_1, y_1, ..., x_n, y_n) = P(x_1|y_1) \prod_{i=2}^{n} P(y_i|y_{i-1}) P(x_i|y_i)$$

If we want to determine a hidden Markov model $\mathcal{M}$, then the following parameters are required in addition to the state space $\mathcal{Y}$ and observation space $\mathcal{X}$:

- Hidden state transition probability matrix $\mathbf{A} = [a_{ij}]_{N \times N}$, where $a_{ij} = P(y_{t+1} = s_j | y_t = s_i)$, $1 \le i,j \le N$, which represents the probability of transitions between states in the model.
- Probability matrix of observation $\mathbf{B} = [b_{ij}]_{N \times M}$, where $b_{ij} = P(x_t = o_j | y_t = s_i)$, $1 \le i \le N$, and $1 \le j \le M$, which represents the observed probability from hidden to display states. In other words, b_{ij} represents the probability that the observed value o_j is observed at time t, if the hidden state at this time is s_i.
- Initial distribution $\theta = (\theta_1, ..., \theta_i, ..., \theta_n)$, which is the probability of occurrence of each state at time $t = 0$, where $\theta_i = P(y_1 = s_i)$ for $1 \le i \le N$. In other words, θ_i is the probability that the initial state is s_i.

In practice, one of the basic problems that people often pay attention to HMM is how to learn the optimal model parameters based on the sample set. It can be described as: given the observation sequence $x = \{x_1, x_2, ..., x_n\}$, how to learn the model parameter $\lambda = [\mathbf{A}, \mathbf{B}, \theta]$ to maximize the probability of occurrence of sequence $P(x|\lambda)$? Based on the traces we obtain from the log history, a HMM need to be constructed that can best describe the observed data.

3.3. Learning Probabilistic Model

The most popular approach to constructing a HMM model is the forward-backward algorithm. However, the forward-backward algorithm can only process one trajectory to construct HMM, so multiple trajectories need to be combined for processing. First, we deal with all the traces obtained from the log repository (the number of traces is assumed to be N). For example, if a trace π_i appears a total of m_i times, we use pair (π_i, m_i) to represent them. The corresponding HMM synthesis algorithm is presented in Algorithm 1.

```
Algorithm 1: HMM generation algorithm
Input: A finite set $traces = \{(\pi_1, m_1), (\pi_2, m_2), \ldots, (\pi_n, m_n)\}$
Output: HMM model $H_m$
$\Pi \leftarrow \varepsilon$; //Initialization of $\Pi$ as an empty string;
$N = \sum_{1}^{n} m_i$; //$N$ is the total number of traces
$a \leftarrow random(1, N)$; //$a$ is a random value
for $j \leftarrow 1$ to $N$ do
    if $a \in [1 + \sum_{1}^{k-1} m_k, \sum_{1}^{k} m_k]$ then
        $\Pi \leftarrow \Pi + \pi_k$; //$\pi_k$ is concatenated to $\Pi$
        $a \leftarrow random(1, N)$;
$H_m = hmm(\Pi)$; // Call forward-backward algorithm;
return $H_m$;
```

Merging all the traces directly will introduce errors into the final model and non-existent paths, such as those transitions from the end states to the initial states. In this case, we add two states to all resulting traces, namely, $S_{init}$and S_{end}. The introduction of these two states will improve the accuracy

```
M = 6
N = 6
A:
0.001342 0.509898 0.487121 0.001781 0.001891 0.001989
0.001897 0.204231 0.275123 0.098343 0.423153 0.001176
0.001279 0.099123 0.281981 0.523891 0.095871 0.001891
0.001871 0.001898 0.001891 0.996891 0.001090 0.000123
0.000671 0.000211 0.001452 0.000983 0.998981 0.001981
0.001981 0.000781 0.000981 0.499901 0.499891 0.001771
B:
0.991842 0.001901 0.001981 0.001075 0.001981 0.001826
0.001781 0.991782 0.001092 0.001981 0.001887 0.001912
0.001089 0.001891 0.991091 0.001892 0.001891 0.001882
0.001901 0.001125 0.001091 0.989012 0.001192 0.001891
0.001193 0.001898 0.001098 0.008915 0.971881 0.001882
0.001183 0.001121 0.001781 0.009811 0.001982 0.991981
θ:
0.999986 0.001003 0.001003 0.001003 0.001003 0.001003
```

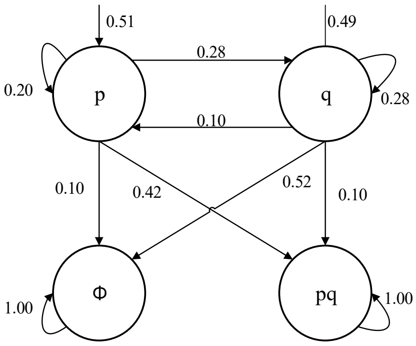


Figure 5. The DTMC (Right) corresponding to the learned HMM (Left,Note that the matrix **A** contains extra state S_{init} and S_{end}.), and the precision of data in DTMC is rounded to two decimals.

of the final model [18]. The classical forward-backward algorithm does not implement the incremental process, fortunately, there is a lot of research to solve the problem, such as the work in [19].

For example in Figure 2, we simulate the process of generating DTMC in UAV platform Ardupilot [20]. The data of all control commands and sensors are recorded into the log system of Ardupilot. We implemented an event parser to obtain concerned events from logs. After simulating the flight for 10000 times, the corresponding traces are extracted and used to generate HMM with Algorithm 1. Left part in Figure 5 depicts the HMM $[\mathbf{A}, \mathbf{B}, \theta]$ learned from 10000 traces, and the graphical representation of its DTMC only contain p and q is shown in the right part of Figure 5. These logs are obtained by monitoring the target drone and its environment. The DTMC model can be obtained from matrix **A** in HMM directly. Through experiments, we find that the accuracy of the forward-backward algorithm is closely related to two factors. First, a longer observation sequence corresponds to a higher accuracy. Second, The similarity between the initial model and the target model will directly affect the efficiency of the algorithm.

4. Probabilistic Monitor Generation

The generated DTMC is the model of system and environment learned, which should be used together with the monitored property to determine and predict the satisfaction of property in specific state. Thus, we will generate a probabilistic monitor from DTMC and the automaton corresponding to the property. Figure 6 shows the framework of the procedures for generating probabilistic monitor.

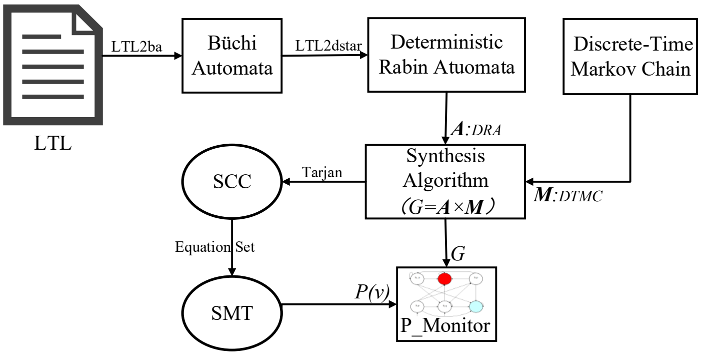

LTL
LTL2ba
Büchi Automata
LTL2dstar
Deterministic Rabin Atuomata
Discrete-Time Markov Chain
A:DRA
Synthesis Algorithm (G=A×M)
M:DTMC
Tarjan
SCC
Equation Set
SMT
P(v)
G
P_Monitor

Figure 6. The framework of probabilistic monitor generation.

ω-automata

A ω*-automaton* $A = (Q, \Sigma, \delta, Q_0, Acc)$ includes the following elements:

- Q is a finite set of states;
- Σ is a finite set called alphabet;

- $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function;
- $Q_0 \subseteq Q$ is the initial state set;
- *Acc* is the acceptance condition.

ω-automata can be classified into deterministic and non-deterministic ones, which mainly differ in transition functions. An automaton is said *deterministic* if $|Q_0| = 1$ and $|\delta(q,a)| = 1$ for each $q \in Q$ and $a \in \Sigma$. In this case, we consider the transition function δ to be of the type $Q \times \Sigma \rightarrow Q$.

An input for A is an infinite string over the alphabet Σ, i.e. it is an infinite sequence $\alpha = a_0, a_1, a_2, ...$ The run of A on such input is an infinite sequence $\rho = r_0, r_1, r_2, ...$ of states, defined as follows:$r_0 \in Q_0$ and $r_i \in \delta(r_{i-1}, a_{i-1})$ where $i \geq 1$. Let ρ denote a run of A over the infinite word $\sigma \in \Sigma^\omega$, and $Inf(\rho)$ be the set of states occurring infinitely in ρ. Given the different acceptance conditions, ω-automata can be classified into different types such as:

- *Büchi automata (BA)*: For accept state set $F \subseteq Q$, $F \cap Inf(\rho) \neq \emptyset$;
- *Rabin automata (RA)*: For the set of pairs $\{(E_1, F_1), (E_2, F_2), ..., (E_m, F_m)\}$, where $E_i, F_i \subseteq Q$, there exists some $1 \leqslant i \leqslant m$ that $E_i \cap Inf(\rho) = \emptyset$ and $F_i \cap Inf(\rho) \neq \emptyset$.

In this paper, we use NBA and DRA to designate nondeterministic Büchi automata and deterministic Rabin automata, respectively.

Directed Graph and SCC

A directed graph G is a tuple (V, E), where:

- V is a finite non-empty set of vertices;
- $E \subseteq V \times V$ is a finite set of directed edges.

Give a directed graph $G = (V, E)$, and $V' \subseteq V$. If for each pair of vertices $x, y \subseteq V'$ there are $x \rightarrow y$ and $y \rightarrow x$, then we call the subgraph of G composed by the node set V' as strongly connected component $SCC\ V'$.

Maximal SCC, Bottom SCC_s and Intermediate SCC_s

If $SCC\ V' \subseteq C \Rightarrow V' = C$ for each $SCC\ C$ of G, then we consider $SCC\ V'$ as maximal SCC and denote it by $SCC_s\ V'$.

If a $SCC_s\ V'$ exists in $G = (V, E)$ and $C = V \setminus V'$, for each vertex $x \in V'$ and $y \in C$, there is no $x \rightarrow y$ holds. We then call it Bottom SCC_s, denoted as $BSCC_s\ V'$.

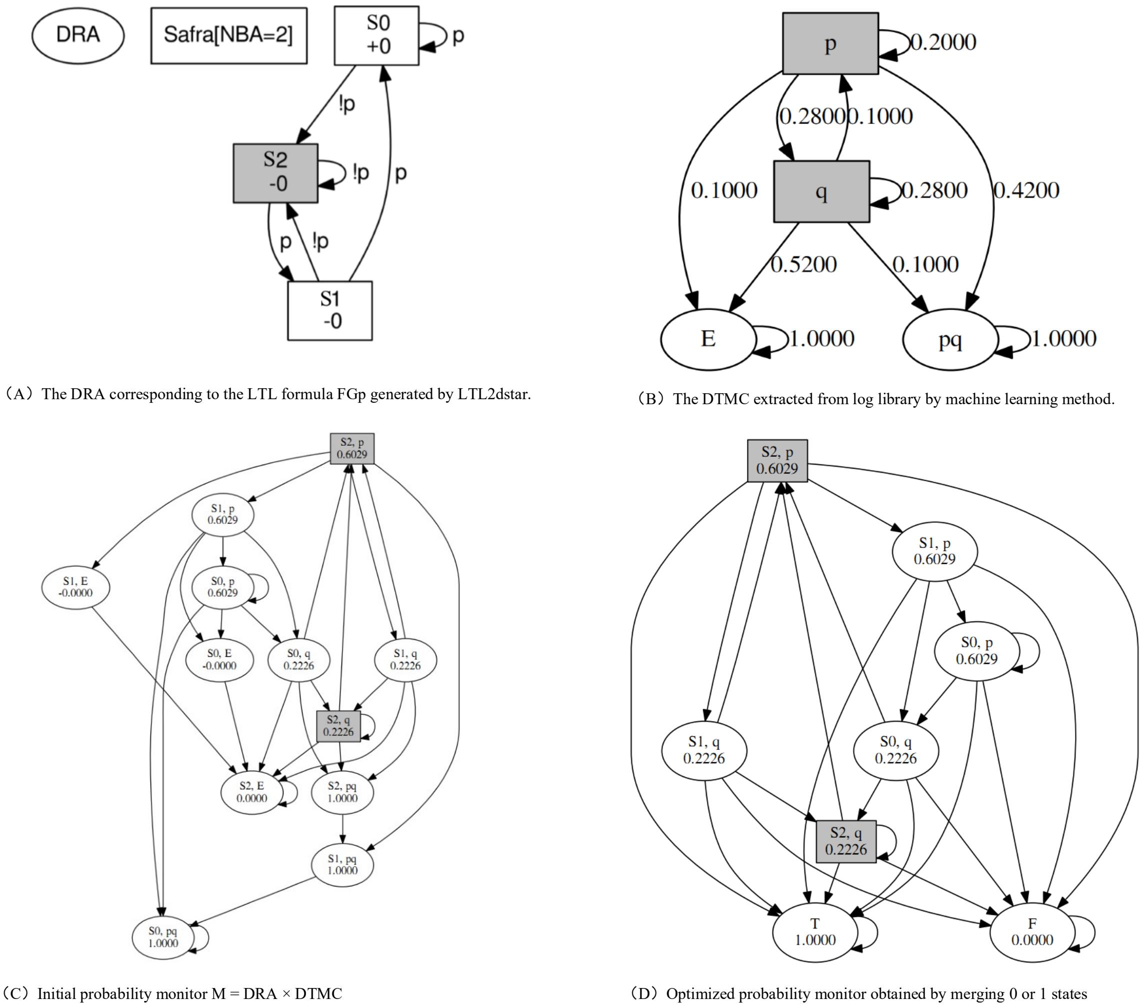
If node set V' is SCC_s, but it is not $BSCC_s$, we say the V' is an Intermediate SCC_s, denoted as $MSCC_s$.

Then given the learned DTMC and the LTL property to be monitored, we will generate probabilistic monitor according to the process shown in Figure 6.

4.1. From LTL to DRA

Various methods of translating a LTL formula into an equivalent Büchi automaton have been proposed in literatures. Unlike model checking, here we focus on the probability of property satisfaction when a new event is observed, thus we hope the monitor should be deterministic. Since deterministic Büchi automata are strictly less expressive than the non-deterministic ones, currently there is no way for translating NBAs into DBAs. But, McNaughton's Theorem and Safra's construction provide the algorithm that can translate a Büchi automaton into a deterministic Rabin automaton [21]. Thus we will use DRA as the automaton formalism of LTL property, which still need BA as intermediate representation.

The tool LTL2BA [22] can convert LTL formulas into Büchi automata. The conversion is implemented in three steps. Firstly, the LTL formula is transformed into a very weak alternating automaton (AWAA). This step mainly deals with the logical relations among the sub-formulas. Secondly, the VWAA is converted into a generalized Büchi automaton (GBA), which mainly deals with



（A）The DRA corresponding to the LTL formula FGp generated by LTL2dstar.

（B）The DTMC extracted from log library by machine learning method.

（C）Initial probability monitor M = DRA × DTMC

（D）Optimized probability monitor obtained by merging 0 or 1 states

Figure 7. Generating probability monitor for property φ of UAS.

the problem of state combinations. Thirdly, GBA is converted into BA by using an easy-to-use and well-known construction method. Finally, we use tool LTL2dstar [23] to convert BA into DRA with a worst-case complexity of $2^{O(n \log n)}$, where n denotes the number of states in BA.

4.2. Generating Probabilistic Runtime Monitor

With the DTMC learned and the DRA constructed from property, we now can generate the probabilistic monitor as follow.

Step1: Product of DRA and DTMC. Given a DRA $A = (Q, \Sigma, \delta, q_0, ACC)$ and a DTMC $M = (S, I, T, L)$, where $\Sigma = S$, we firstly construct their production. Define the directed graph $\mathcal{G} = A \times M = (V, E)$, where $V = \{(q, s) | q \in Q, s \in S\}$. An edge $e \in E$ has the form $e = ((q, s), (q', s'))$, which should satisfy $T(s, s') > 0$ and $q' = \delta(q, s)$. Figure 7 presents an example of this construction process, where the DRA is generated by the LTL formula $\varphi = FGp$ which is a property should be satisfied by the case in Figure 2.

Step2: Probability Calculation. We use $P(v)$ to denote the probability that node $v = (q, s) \in V$ will satisfy the LTL property in finite graph $\mathcal{G}$. In order to calculate the probability of each node in $\mathcal{G}$, the concept of $Accepted\_SCC_s$ is introduced. The acceptance condition of the DRA generated by LTL is $ACC = \{(E_1, F_1), ..., (E_k, F_k)\}$. A $BSCC_s$ C is *accepted* if there is some i such that $C \cap E_i = \emptyset$ and $C \cap F_i \neq \emptyset$. Otherwise the $BSCC_s$ C is *rejected*. For node $v = (q, s)$ in SCC_s C, its probability will be obtained as following:

(1) If C is an $Accepted\_SCC_s$ then $P(v) = 1$;

(2) If C is an $Rejected\_SCC_s$ then $P(v) = 0$;

(3) If C is an $Intermediate\_SCC_s$ then $P(v) = \sum_{u \in W}(T(s,s') \times P(u))$, W is set of all successors of v, and u is denoted as (q',s') where $q' = \delta(q,s)$.

We use the case in Figure 7 to describe the procedures. Figure 7A and 7B present the DRA and DTMC which are input of algorithm 1. Figure 7C gives the product of them. The initial states are nodes (S_2, p) and (S_2, q), then we can obtain all SCC_s using the classic Tarjan algorithm and we name it $FindSCC_s$. As shown in Figure 7C, except for nodes (S_2, p), (S_2, q), (S_1, q), (S_1, p), (S_0, p), (S_0, q) that make up one SCC_s, every other node is a SCC_s.

According to the previous definition, we can easily know that node (S_0, pq) and node $(S_2, \emptyset)$ are $BSCC_s$. Based on the definition of $Accepted\_BSCC_s$ and the DRA structure, we can work out the probability of $Accepted\_BSCC_s$ $(S0, pq)$ with above (1) and $Rejected\_BSCC_s$ $(S2, \emptyset)$ with (2). For all $MSCC_s$, we obtain P by solving the equations based on (3). For instance, we can calculate the probability of nodes in one $MSCC_s$ with following equations:

$$
\begin{aligned}
P(S_0,\ p) &= P(S_0,p) * T(p,p) + P(S_0,pq) * T(p,pq) \\
&\quad + P(S_2,q) * T(p,q) + P(S_2,\emptyset) * T(p,\emptyset) \\
P(S_0,\ q) &= P(S_2,p) * T(q,p) + P(S_2,\emptyset) * T(q,\emptyset) \\
&\quad + P(S_2,pq) * T(q,pq) + P(S_2,q) * T(q,q) \\
P(S_1,\ p) &= P(S_0,p) * T(p,p) + P(S_0,pq) * T(p,pq) \\
&\quad + P(S_0,\emptyset) * T(p,\emptyset) + P(S_0,q) \\
P(S_1,\ q) &= P(S_2,p) * T(q,p) + P(S_2,q) * T(q,q) \\
&\quad + P(S_2,\emptyset) * T(q,\emptyset) + P(S_2,pq) * T(q,pq) \\
P(S_2,\ p) &= P(S_1,p) * T(p,p) + P(S_1,pq) * T(p,pq) \\
&\quad + P(S_2,q) * T(p,q) + P(S_2,\emptyset) * T(p,\emptyset) \\
P(S_2,\ q) &= P(S_1,pq) * T(q,pq) + P(S_1,p) * T(q,p) \\
&\quad + P(S_2,q) * T(q,q) + P(S_2,\emptyset) * T(q,\emptyset)
\end{aligned}
$$

The above equations are solved using $Gaussian-\ Elimination$ method and the probability of each node can be obtained then. As show in these equations, to calculate the probability of node (S_2, q), we should know the probability of node $(S_2, \emptyset)$ which we have worked out before. So in fact the process of the calculation is Depth-First-Search (DFS) which consists with the process of Tarjan algorithm. Because of this fact, once we find a SCC_s through the Tarjan algorithm, we can calculate the probability of each node in this SCC_s which meet formula $\varphi = FGp$, as shown in Figure 7C.

Step3: Optimization. After calculating the probability of each state based on $\mathcal{G} = A \times M$, $\mathcal{G}$ and the labelled probability make up the runtime monitor. For some states in the monitor, the probability labels are 0 or 1. These states can be merged to optimize the structure of the monitor. In this example, given that the probability labels for states (S_0, pq) and (S_1, pq) are both 1, they can be merged to optimize the monitor as shown in Figure 7D.

5. Implementation and Evaluation

The tool framework and running process of the probabilistic runtime monitor is briefly described in Figure 8. It extracts the events of interest from event acquirer module. When an event occurs, the probabilistic monitor will perform a transition based on its current state and the recieved event. Based on the system requirements, if the probability value exceeds our specified gate value, then some predefined behavior will be executed, such as an alarm. When an alarm occurs in the system, the events labelled on the transitions in the monitor following current state can help controller to change the direction of the system running by trigger a specific event.

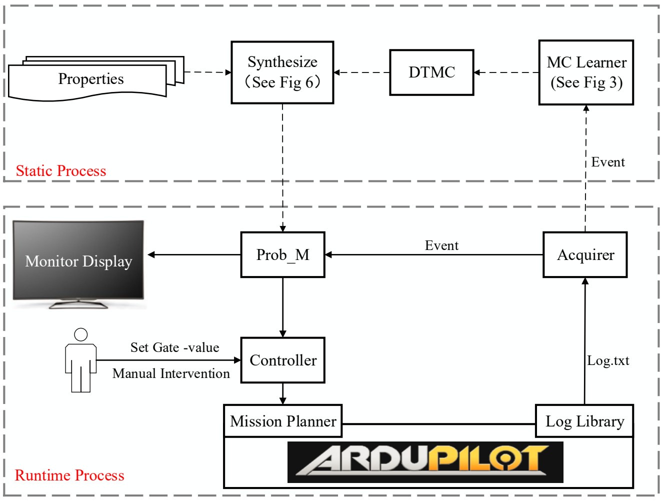


Figure 8. The framework of tool implementation in the UAS simulation platform Ardupilot.

In the process of running the system, the *Acquirer* obtains the events from the target system and environment, and *Prob_M* is the main part of the monitor platform which stores the structure of the monitor and changes its state based on the event obtained by Acquirer. The *Controller* decides which control event should be issued in next step when the system encounters a problem, or prompts what external command intervention are needed to maintain the system in safe state. In the static phase, we can use the traces newly added to the log library to update the learned model, which can continuously improve the accuracy of the probabilistic monitor.

Compared with our work, traditional RV monitor under binary semantics of LTL faces the semantic problem of consistency, since they should give a new semantics of LTL on finite traces. Although the monitoring method under three-valued semantics of LTL can satisfy impartial and anticipatory requirements [7], the monitor does not produce more quantitative results when the satisfaction of property in many states are output as "?" (*inconclusive*). Meanwhile, the probabilistic monitor can exactly determine the quantitative evaluation of satisfying the LTL property in current state, thereby greatly expanding its application scenarios and making up the insufficiency of other RV methods.

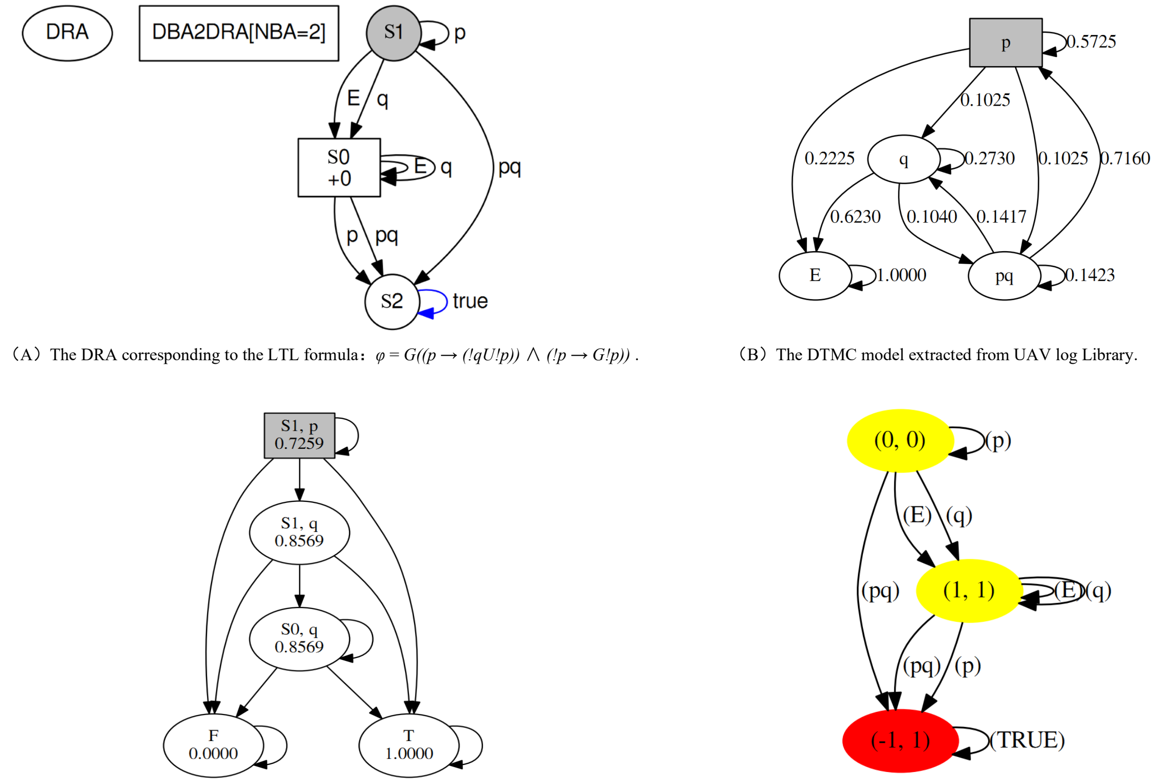
Furthermore, our probabilistic monitors can adjust the subsequent execution of the target system. Specifically, when the system is running, people know which “event" can increase or decrease the future probability for the system to meet the property. The existing runtime monitors are unable to achieve this purpose.

To show the effectiveness of the method proposed above, we applied the probabilistic RV and its tool to actual UAS platform Ardupilot. It is generally known that the hardware or software defects and the presence of external malicious attacks pose a great threat to the security of UAS. Let’s consider the following situation.

When UAS reconnaissances a hostile force area, if it encounters dangerous conditions (such as abnormal signal interference), in order to balance the task completion and UAS safety, we set the following rules: Firstly, if detection task has been initialized, the drone will not accept abnormal signal (both from the region and the console), until the task has finished or the task is interrupted (such as receiving a normal command of stopping detection); Secondly, if the task has not been initialized after enter this hostile area, the task will not be started to ensure the safety of UAS.

Therefore, we define the following events: (1) p: the UAS is in the state of executing critical task, and (2) q: the UAS has received abnormal instruction. Then the property can be expressed by LTL formula:

$$\varphi = G((p \to (\neg q U \neg p)) \wedge (\neg p \to G\neg p))$$



（A）The DRA corresponding to the LTL formula：$\varphi = G((p \to (!qU!p)) \wedge (!p \to G!p))$.

（B）The DTMC model extracted from UAV log Library.

（C）Probability monitor generated by the algorithm in this paper.

（D）The monitor generated by tool LTL3tools is based on three-value semantics of LTL.

Figure 9. The experiments of probability monitor compared with monitor based on three-value semantics.

The corresponding DRA of property φ is generated and shown in figure 9A, and the DTMC model (based on propositions p and q) of this UAS is obtained by learning the log history of multiple flight simulations, which is shown in figure 9B. The probabilistic monitor is generated from above DRA and DTMC as shown in figure 9C. Depend on this monitor, 200 system running traces with length 1000 (in fact the trace should be infinite, here length 1000 can embody the effectiveness of the method) are checked whether they satisfied this safety property. As the result, there are 140 traces finally arrive the state with probability 1.000 that satisfies the property, and for the left 60 traces the probability is 0.000. The ratio of the traces satisfying the property in all traces is 70%, which is close to the probability on the initial node with the probability 72.59%.

To compare with our method, similar experiments with same data is conducted under the semantics of LTL_3. Firstly, the same LTL formula is transformed into the monitor shown in figure **??** D by applying LTL_3 tool. Obviously, there is no state of *true* in this monitor because the formula is begin with operator G, and *true* cannot be obtained in infinite word under the semantics of LTL_3. It is because LTL_3 method does not consider the environment that system runs in. But our probabilistic monitor learned the model from history of both system and environment, and use these information in the monitor. By repeating the same 200 traces, the LTL_3 monitor get the results with 140 uncertain and 60 false. It can be found that probabilistic monitor can get not only the quantitative value of system's current safety status, but also the probability when it is running in uncertain environment, which takes the advantage of Markov chain.

6. Conclusion

For the system without accurate model and program code, or running in the uncertain environment that threaten cannot be predicted before deployment, we propose an approach to constructing probabilistic monitor to detect property violation at runtime. The method utilizes the history traces from the log repository of target system to learn the HMM model which represents the behavior of both system and environment. Then the DTMC model is obtained from HMM. Together with the DRA generated from LTL formula, the probabilistic runtime monitor can be generated after calculating the probability of state in the product of DTMC and DRA. Probabilistic monitor has a good application scenario in which the LTL property originally cannot be quantitatively judged with existing RV methods, and it also can provide directional guidance for system intervention. We implemented the corresponding tool on the UAS platform Ardupilot, and the experiments show the effectiveness comparing with other methods such as LTL_3.

In the future work, we plan to study the method of building probabilistic monitor with online incrementally learning, so that when the target system's log library grows, the model learned can become more accurate in time. Furthermore, for the probabilistic monitor containing guidance information, we will study an effective intervention mechanism.

References

1. Zhang, P.; Su, Z.; Zhu, Y.; Li, W.; Li, B. WS-PSC Monitor: A Tool Chain for Monitoring Temporal and Timing Properties in Composite Service Based on Property Sequence Chart. International Conference, 2010, pp. 485–489.
2. Electrical, I.O.; Board, I.S. IEEE Standard for Software Verification and Validation. *Software Quality Professional* **2005**, pp. 1–217.
3. Clarke, E.M. Model Checking-My 27-Year Quest to Overcome the State Explosion Problem. Logic in Computer Science, 2009. LICS '09. IEEE Symposium on, 2008, pp. 3–3.
4. Dahl, O.J.; Dijkstra, E.W.; Hoare, C.A.R. *Structured programming*; Academic Press, 1972; pp. 179–185.
5. Zhao, C.; Dong, W.; Wang, J.; Sui, P.; Qi, Z. Software active online Monitoring under anticipatory Semantics. *Shm* **2009**.
6. Yu, K.; Chen, Z.; Dong, W. A Predictive Runtime Verification Framework for Cyber-Physical Systems. IEEE Eighth International Conference on Software Security and Reliability-Companion, 2014, pp. 247–250.
7. Bauer, A.; Leucker, M.; Schallhart, C. Comparing LTL Semantics for Runtime Verification. *Journal of Logic & Computation* **2010**, *20*, 651–674.
8. Naskos, A.; Stachtiari, E.; Katsaros, P.; Gounaris, A. *Probabilistic Model Checking at Runtime for the Provisioning of Cloud Resources*; Springer International Publishing, 2015; pp. 275–280.
9. Filieri, A.; Tamburrelli, G. *Probabilistic Verification at Runtime for Self-Adaptive Systems*; 2013; pp. 30–59.
10. Nouri, A.; Raman, B.; Bozga, M.; Legay, A.; Bensalem, S. Faster Statistical Model Checking by Means of Abstraction and Learning. Runtime Verification, 2014, pp. 340–355.
11. Sammapun, U.; Lee, I.; Sokolsky, O.; Regehr, J. *Statistical Runtime Checking of Probabilistic Properties*; Springer Berlin Heidelberg, 2007; pp. 164–175.
12. Ngo, V.C.; Legay, A.; Joloboff, V. PSCV: A Runtime Verification Tool for Probabilistic SystemC Models **2016**.
13. Jayaputera, J.; Poernomo, I.; Schmidt, H. Runtime Verification of Timing and Probabilistic Properties using WMI and .NET. Euromicro Conference, 2004. Proceedings., 2004, pp. 100–106.
14. Zhao, C.; Dong, W.; Qi, Z. Active Monitoring for Control Systems under Anticipatory Semantics. International Conference on Quality Software, 2010, pp. 318–325.
15. Chen, Z.; Wei, O.; Huang, Z.; Xi, H. Formal Semantics of Runtime Monitoring, Verification, Enforcement and Control. International Symposium on Theoretical Aspects of Software Engineering, 2015, pp. 63–70.
16. Giannakopoulou, D.; Havelund, K. Runtime Analysis of Linear Temporal Logic Specifications. 2001.

17. Chu, S.M.; Huang, T.S. An experimental study of coupled hidden Markov models. IEEE International Conference on Acoustics, Speech, and Signal Processing, 2002, pp. IV–4100–IV–4103.
18. Abbasi, N.M. Hidden Markov Methods. Algorithms and Implementation **2015**.
19. Motik, B.; Nenov, Y.; Piro, R.; Horrocks, I. Incremental update of datalog materialisation: the backward/forward algorithm. Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015, pp. 1560–1568.
20. An simulation platform of unmanned aerial vehicle. http://www.ardupilot.org/.
21. Safra, S. On the complexity of ω-automata. Foundations of Computer Science, 1988., Symposium on, 1988, pp. 319–327.
22. Gastin, P.; Oddoux, D. Fast LTL to Büchi Automata Translation. International Conference on Computer Aided Verification, 2001, pp. 53–65.
23. LTL to deterministic Streett and Rabin automata. http://www.ltl2dstar.de/.