Statistical Runtime Checking of Probabilistic Properties

Usa Sammapun[1], Insup Lee[1], Oleg Sokolsky[1], and John Regehr[2]

[1] University of Pennsylvania, {usa,lee,sokolsky}@cis.upenn.edu
[2] University of Utah, regehr@cs.utah.edu

Abstract. *Probabilistic correctness is an important aspect of reliable systems. A soft real-time system, for instance, may be designed to tolerate some degree of deadline misses under a threshold. Since probabilistic systems may behave differently from their probabilistic models depending on their current environments, checking the systems at runtime can provide another level of assurance for their probabilistic correctness. This paper presents a statistical runtime verification for probabilistic properties using statistical analysis. However, while this statistical analysis collects a number of execution paths as samples to check probabilistic properties within some certain error bounds, runtime verification can only produce one single sample. This paper provides a technique to produce such a number of samples and applies this methodology to check probabilistic properties in wireless sensor network applications.*

Key words: Runtime verification, statistical monitoring, probabilistic properties

1 Introduction

Probabilistic correctness is an important aspect of reliable systems, which could tolerate some undesirable behaviors such as deadline misses or data loss. For example, unlike hard real-time systems that strictly require computation to complete within its deadline, soft real-time systems can tolerate some degree of deadline misses. We can characterize this degree in terms of the acceptable probability of a deadline miss. Another example is a wireless sensor network application with probabilistic constraints on its behaviors to tolerate some degree of data loss. Since probabilistic systems may deviate from their probabilistic requirements due to unexpected environments or incorrect implementation, checking the systems at runtime in addition to a static probabilistic check can provide additional level of assurance for their probabilistic correctness.

Runtime verification is a technique for checking correctness of a system at runtime by observing a system execution and checking it against its property specification. One runtime verification framework is called MaC or Monitoring and Checking [9, 12]. MaC provides expressive specification languages based on Linear Temporal Logic [11] to specify system properties. Once the properties are specified, MaC observes the system by retrieving system information from probes instrumented into the system prior to the execution. MaC then checks the execution against the system properties and reports any violations.

To check probabilistic properties, runtime verification can adopt the statistical technique [16] used in model checking to verify probabilistic properties. The statistical technique simulates, samples many execution paths, and estimates probabilities by counting successful samples against all samples. After the probabilities are estimated, statistical analysis such as hypothesis testing is used to determine statistically whether a system satisfies a probabilistic property with a given level of confidence.

One particular difficulty in using this technique in runtime verification, however, is that a runtime checker follows only one execution path and cannot easily collect many different executione paths as in probabilistic model checking. Therefore, this one execution path, usually in a form of a trace of states or events, needs to be decomposed into different individual samples, which can be done only if a probabilistic system being observed has repeated or periodic behaviors. Such behaviors are typically exhibited by the systems in our target domain. Soft real-time schedulers repeatedly schedule tasks; network protocols repeatedly transmit or receive messages. This paper describes how MaC can break down one execution into different individual samples and how MaC adopts the statistical technique to check probabilistic properties at runtime. This technique has been applied to check probabilistic properties in wireless sensor network applications.

Our contributions are: 1) we provide a general statistical technique for checking probabilistic properties at runtime, 2) the technique is described to and implemented in an existing runtime verification framework called MaC, and 3) a case study is presented for checking probabilistic properties in wireless sensor network application.

Related Work. Runtime verification frameworks based on Linear Temporal Logic, such as Java PathExplorer [7], work by Kristoffersen *et al.* [10], and work by Stolz and Bodden [13], typically cannot be used to check probabilistic properties. Those that provide probabilistic properties such as Eagle [1], Temporal Rover [4], and a framework by Jayaputera et al. [8] do not prescribe statistical analysis to support the estimated probabilities. Finkbeiner *et al.* [5] discussed collection of statistics over execution traces, yet their work was not concerned with probability estimation.

2 Background: MaC

Monitoring and Checking or MaC [9, 12] is an established runtime verification framework that can be used to check whether that a program is executing correctly with respect to its formal requirement specification. Before execution, specification is written, and a program is instrumented with probes to extract observation. During runtime, a program execution is observed and checked against the formal specification. An *event recognizer* detects low-level observation specific to program implementation and transforms it into high-level information, which is forwarded to a *checker*. A *checker* determines whether the high-level information satisfies the formal specification. If violations are detected, the checker reports to the user.

The main aspect of MaC is the formal requirement specification. MaC provides two specification languages. The low-level monitoring specification or Primitive Event Definition Language (PEDL), defines which low-level application-dependent observation is extracted, and how the observation is transformed into high-level information. The high-level requirement specification or Meta-Event Definition Language (MEDL), based on Linear Temporal Logic (LTL) [11], allows one to specify safety properties in terms of high-level information. PEDL is tied to a particular implementation while MEDL is independent of any implementation. Only MEDL is presented in this paper. See MaC [9] for PEDL.

High-level information in MEDL can be distinguished into events or conditions. Events occur instantaneously during execution, whereas conditions represent system states that hold for a duration of time and can be *true*, *false*, or *undefined*. For example, an event denoting a call to a method *init* occurs at the instant the control is passed to the method, and a condition $v < 5$ holds as long as the value v is less than 5. Events and conditions can be composed using boolean operators such as negation !, conjunction &&, disjunction ||, and other operators, as shown in Fig. 1.

$$\begin{array}{l} E ::= e \mid E || E \mid E \&\& E \mid start(C) \mid end(C) \mid E\ when\ C \\ C ::= c \mid !C \mid C || C \mid C \&\& C \mid C \rightarrow C \mid defined(C) \mid [E, E) \end{array}$$

Fig. 1. Syntax of events and conditions

There are some natural events associated with conditions, namely, an instant when a condition C becomes *true* and *false*, denoted as $start(C)$ and $end(C)$, respectively. An event $(E\ when\ C)$ is present if E occurs at a time when a condition C is *true*. A condition $defined(C)$ is true whenever a condition C has a well-defined value, namely, *true* or *false*. Any pair of events define an interval forming a condition $[E_1, E_2)$ that is *true* from an event E_1 *until* an event E_2.

MEDL distinguishes special events and conditions that denote system specification. Safety properties are conditions that must *always* be true during an execution. Alarms, on the other hand, are events that must *never* be raised. From the viewpoint of expressiveness, both safety properties and alarms correspond to the safety properties [11].

3 Probabilistic Properties

MaC offers additional syntax and semantics for specifying probabilistic properties. To check these probabilistic properties, MaC adopts a statistical technique used in model checking [16]. The statistical technique simulates, samples many execution paths, and estimates probabilities by counting successful samples against all samples. MaC and other runtime verification frameworks operate on the current execution path and are not typically designed to accumulate data from many different execution paths. Because of this, the current execution path, usually in a form of a trace of states or events, needs to be decomposed into non-overlapping individual samples, which can be done only if a probabilistic system being observed has repeated or periodic behaviors such as soft real-time schedulers or network protocols.

To decompose an execution, MaC distinguishes one repetitive behavior from another by using conditional probabilities. Written in terms of probabilistic properties, one can specify as given a condition A, does the probability that an outcome B occurs fall within a given range? In terms of MaC events, one can specify as given that an event e_0 occurs, does the probability that an event e will occur fall within a given threshold? This way, a sample space is reduced from events of the entire system to only those events relevant to a given probabilistic property. A set of e_0 and e can be collected as one individual sample, and a sequence of these sets can be collected as many different individual samples.

The probability observed from the system can be estimated by counting the outcome event e that occurs in response to the given event e_0 against all the outcome event e. After probabilities are estimated, MaC uses statistical analysis to determine statistically whether a system satisfies a probabilistic property using hypothesis testing. Hypothesis testing provides a systematic procedure with an adequate level of confidence to determine the satisfiability of probabilistic properties.

3.1 Syntax

To specify probabilistic constraints, we extend MaC with a probabilistic event operator. The operator expresses the property that an event e occurs within a certain probability threshold given that an event e_0 occurs. The syntax for the new operator is $e\ pr(\odot p_0, e_0)$ where $\odot \in \{<, >\}$ and p_0 is a probability constant. The event $e\ pr(\odot p_0, e_0)$ can be used in any context where ordinary MaC event operators (listed in Fig. 1) can. This event is raised when the checker has accumulated enough confidence to reject the hypothesis that the above property does not hold.

3.2 Semantics

To give semantics for the probabilistic properties, we describe how samples can be collected from an execution and then show how to use hypothesis testing over this set of samples to determine statistically the satisfaction of probabilistic properties.

Recall that we have to answer the following question: given that an event e_0 occurs, does the probability that an event e will occur fall within a given threshold? Such probabilistic properties can be defined directly using conditional probabilities as

$$Pr(e|e_0) = \frac{Pr(e \text{ and } e_0)}{Pr(e_0)}$$

To estimate $Pr(e \text{ and } e_0)$, let m be the number of occurrences of all MaC events in the trace. Let's call these MaC events *experiments*. Let $X = X_1 + X_2 + ... + X_m$ be the random variable representing the number of successful experiments. Here, X_i is the random variable representing the result of the i^{th} experiment. Then, $X_i = 1$ when the i^{th} experiment is successful, and $X_i = 0$ otherwise. The experiment is successful when e occurs in response to e_0. By this we mean that either e occurs at the same time as e_0, or e follows an occurrence of e_0, without a prior occurrence of e in between. Formally, $X_i = 1$ when the event $e' = e \mathbin{\&\&} (e_0 \mid\mid \mathit{end}([e_0, e]))$ occurs at time t_i, and $X_i = 0$ otherwise. Note that our notion of "occurs in response" does not necessarily imply a causal dependency between the most recent occurrence of e_0 and the current occurrence of e, but reflects the fact that there is an occurrence of e_0 that has not been matched to an earlier occurrence of e.

Therefore, each X_i has a Bernoulli distribution with an unknown parameter $q \in [0, 1]$ where $Pr(X_i = 1) = q$, meaning that the probability that e' will occur is equal to q. X, therefore, has a Binomial distribution with parameters m and q. Finally, let $\bar{q}$ be an observed probability obtained from the samples we collect where $\bar{q} = \frac{X}{m} = \frac{\Sigma X_i}{m}$

Using similar reasoning, let $Y = Y_1 + Y_2 + ... + Y_m$ be a random variable representing the number of e_0 occurrences, where m is the number of all MaC events or *experiments*. $Y_i = 1$ when e_0 occurs at in the i^{th} experiment, and $Y_i = 0$ otherwise. Each Y_i has a Bernoulli distribution with an unknown parameter $q' \in [0, 1]$ where $Pr(Y_i = 1) = q'$. Thus, Y has a Binomial distribution with parameters m and q'. Let $\bar{q'}$ be an observed probability obtained from the samples where $\bar{q'} = \frac{Y}{m} = \frac{\Sigma Y_i}{m}$.

Let $p = Pr(e|e_0)$, and let $\bar{p}$ be an observed probability of p. Since

$$p = Pr(e|e_0) = \frac{Pr(e \text{ and } e_0)}{Pr(e_0)} = \frac{Pr(X_i = 1)}{Pr(Y_i = 1)},$$

then,

$$\bar{p} = \frac{\bar{q}}{\bar{q'}} = \frac{\frac{\Sigma X_i}{m}}{\frac{\Sigma Y_i}{m}} = \frac{\Sigma X_i}{\Sigma Y_i}.$$

Hence, the observed probability $\bar{p}$ is a ratio of the number of occurrences of the event e' over the number of e_0 occurrences:

$$\bar{p} = \frac{|\text{occurrences of } e'|}{|\text{occurrences of } e_0|}$$

e' also ensures that the number of occurrences of e_0 is always greater or equal to the number of occurrences of $e \mathbin{\&\&} (e_0 \mid\mid \mathit{end}([e_0, e]))$, and thus $\bar{p}$ will always be less than or equal to 1. For the rest of this paper, let n be the number of occurrences of e_0.

Hypothesis Testing. Assume one needs to check a probabilistic property $e\ pr(\odot p_0, e_0)$ where $\odot \in \{<, >\}$, and p_0 is a probability bound. It means given that an event e_0 occurs, does the probability that an event e will occur fall within p_0? The observed probability $\bar{p}$ needs to be tested against p_0. This is done by using $\bar{p}$ to approximate the true Binomial probability p based on a statistical procedure of hypothesis testing. The first step, done before running experiments, is to set up two hypotheses H_0 and H_A. H_0, called null hypothesis, is what we have previously believed and what we want to use the hypothesis testing to *disprove*. H_A, called alternative hypothesis, is an alternative to H_0; we will believe H_A only if the data supports it strongly. In our case, we previously believe that the probabilistic event does not occur, and we would trigger the probabilistic event only when we have strong evidence. For example, if the probabilistic event is $e\ pr(< p_0, e_0)$, then H_0 is $p \geq p_0$ and H_A is $p < p_0$. Hence, the acceptance of H_0 means $M, t \not\models e\ pr(< p_0, e_0)$, and the acceptance of H_A means $M, t \models e\ pr(< p_0, e_0)$.

To perform hypothesis testing, we first define our test procedure. A test procedure is the rule for making a decision on whether to accept or reject hypothesis. It has two components: a test statistic and a rejection region. A test statistic is a function of the sample data that is used to decide hypothesis acceptance or rejection. A rejection region is a set of values in which a null hypothesis H_0 would be rejected. Our test statistic is based on the *z-score*, which represents how far a normally distributed sample data is from the population mean. The z-score allows us to tell whether the difference is statistically significant.

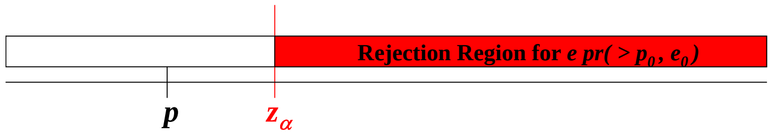
From the Central Limit Theorem, a large number of samples from any distribution approximates Normal distribution. For the Binomial distribution, the sample is considered large enough when it satisfies both $np \geq 10$ and $n(1-p) \geq 10$ [3]. Our implementation ensures that these constraints are satisfied before a decision on the property satisfaction is made. Once the necessary sample size is reached, the *z-score* can be used as our test statistic. The following equation is used to calculate the *z-score* for $\bar{p}$ [3]:

$$Z = \frac{\bar{p} - p_0}{\sqrt{p_0(1-p_0)/n}} \tag{1}$$

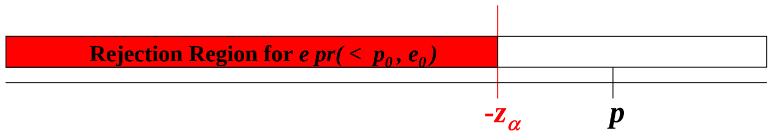
Since $\bar{p} = \frac{X}{Y}$ where X and Y are random variables with Binomial distribution, then Z is a random variable with an approximately standard normal distribution. z is defined as the expected value of Z. Positive values of z mean that $\bar{p}$ is greater than p_0, while negative values mean that it is less than p_0. When z is close to zero, $\bar{p}$ is close to p_0. These simple observation will help us define the rejection region.

To find the rejection region, we utilize the notion of error bounds. There are two kinds of error bounds, known as Type I (α) and Type II (β). Type I error is the probability of incorrectly verifying a property satisfaction and Type II error is the probability of incorrectly verifying a property violation. Formally, $\alpha = Pr\{\text{reject } H_0 | H_0 \text{ is true}\}$ and $\beta = Pr\{\text{accept } H_0 | H_A \text{ is true}\}$. We bound the acceptable Type I error by the *significance level* z_α, and use this bound as the rejection region. For example, if $H_0 : p \leq p_0$ and $H_A : p > p_0$, then

$$\alpha = Pr\{Z \geq z_\alpha \text{ when } Z \text{ has approximately a standard Normal distribution}\}$$



(a) Upper-tailed Test



(b) Lower-tailed Test

Fig. 2. Upper- and Lower-tailed Tests

Since the value of the test statistic can be calculated using (1), hypotheses and rejection regions can be set up, and a decision can be made. Consider $e\ pr(< p_0, e_0)$ and $e\ pr(> p_0, e_0)$.

(a) **Upper-tailed test: $e\ pr(> p_0, e_0)$.** A hypothesis is set up as $H_0 : p \leq p_0$ and $H_A : p > p_0$. The rejection region is $z \geq z_\alpha$, shown in Fig. 2 (a). Thus, reject H_0 if $z \geq z_\alpha$, meaning there is strong evidence supporting that $\bar{p}$ is greater than p_0. Accept H_0 otherwise. Hence, an event $e\ pr(> p_0, e_0)$ is raised or $M, t \models e\ pr(> p_0, e_0)$ when H_0 is rejected because of the strong evidence supporting that the probability of e occurring given e_0 is greater than p_0.

(b) **Lower-tailed test: $e\ pr(< p_0, e_0)$.** A hypothesis is set up as $H_0 : p \geq p_0$ and $H_A : p < p_0$. The rejection region is $z \leq -z_\alpha$, shown in Fig. 2 (b). Thus, reject H_0 if $z \leq -z_\alpha$, meaning there is strong evidence supporting that $\bar{p}$ is less than p_0. Accept H_0 otherwise. Hence, an event $e\ pr(< p_0, e_0)$ is triggered or $M, t \models e\ pr(< p_0, e_0)$ when H_0 is rejected because of the strong evidence supporting that the probability of e occurring given e_0 is less than p_0.

The two error types have an inverse effect on each other: decreasing the value of α will increase the value of β. The value of β depends on the true value of a system's probability p. Assuming that $p = p'$, for the upper-tailed test β is the following function of p': $\beta(p') = Pr\{Z < z_\alpha \text{when } p = p'\}$, and can be estimated as

$$\beta(p') = \Phi\left(\frac{p_0 - p' + z_\alpha\sqrt{p_0(1-p_0)/n}}{\sqrt{p'(1-p')/n}}\right).$$

Similarly, for the lower-tailed test, $\beta(p')$ is $Pr\{Z > -z_\alpha \text{when } p = p'\}$. Thus,

$$\beta(p') = 1 - \Phi\left(\frac{p_0 - p' + z_\alpha\sqrt{p_0(1-p_0)/n}}{\sqrt{p'(1-p')/n}}.\right)$$

3.3 Discussion

The implemention for checking probabilistic properties is done through a sliding window technique. When the number of experiments is small, both Type I and Type II errors can be large. When Type I error or α is fixed and as the number of experiments increase, Type II error or β decreases providing more reliable results. Thus, more samples or more experiments can increase the confidence in the results. However, in our first implementation of hypothesis testing [12], we noticed that considering too many experiments leads to false alarms.

Recall from the semantics that the z value depends directly on the number of experiments n. When p_0 and the observed probability $\bar{p}$ are fixed, if n increases, z also increases and can become very sensitive to p_0. It means that when $\bar{p}$ only differs slightly from p_0, an alarm can be triggered. In practice, the observed probability that is only slightly different from p_0 has little practical significance while the observed probability that differs from p_0 by a large magnitude would be worth being detected. Thus, our goal is to detect only those behaviors that greatly differ from the desired probabilistic behaviors.

Consider the miss deadline example specified as $missDeadline\ pr(> 0.2, startT)$. If $|\,missDeadline\ \&\&\ (startT\ ||\ \ end([startT, missDeadline)\,|$ is 21 and $|\,startT\,|$ events is 100, then the observed $\bar{p} = \frac{21}{100} = 0.21$ and its z-score is $z = 0.25$. With $\alpha = 97.5\%$ and $z_\alpha = 1.96$, then $z < z_\alpha$, which provides no strong evidence that the observed probability $\bar{p} = 0.21$ is greater than $p_0 = 0.2$. Therefore, an alarm is not raised. However, with the same observed probability $\bar{p} = 0.21$ where $n = 10000$ and $|\ missDeadline\ \&\&\ (startT\ ||\ end([startT, missDeadline)\ | = 2100$, its z-score is $z = 2.50$. Assuming the same $\alpha = 97.5\%$ and $z_\alpha = 1.96$, then $z \geq z_\alpha$, meaning that there is strong evidence that the observed probability $\bar{p} = 0.21$ is greater than $p_0 = 0.2$, and thus, an alarm is raised. This example shows that the same observed probability and error bounds can produce different hypothesis testing decisions depending on the number of experiments considered. It follows that although a higher number of experiments can provide higher confidence of detecting violations, it also generates false alarms.

This effect is studied in the area of statistics known as sequential analysis [15]. The proposed solution is to adjust z_α when more samples become available. Here, we provide an alternative solution that keeps the significance level constant but instead removed older samples from the set. We maintain the sliding window of samples that keeps the number of experiments used in checking constant. When the number of observed experiments exceeds the window size, we discard the earliest experiments. In MaC, the default window size is chosen to satisfy the constraints $np \geq 10$ and $n(1-p) \geq 10$. This way, we can ensure that the sample size is large enough to approximate Normal distribution and reduce Type

II errors but still not too large to affect the sensitivity of the *z-score* to large sample size. We believe that the size of the window should also depend on the chosen significance level. This relation is the subject of our on-going research.

Sliding windows also help deal with unobservable mode switches in the system. For example, a soft real-time task may miss its deadline more often in an emergency situation, when a machine is more heavily loaded with extra tasks to handle the emergency, than in the nominal case, when the machine is more lightly loaded. Without the sliding window, experiments that occurred before the mode switch would affect the statistics long after the mode switch happens and delay — or even prevent — the detection of the probabilistic property violation in the new mode. The underlying assumption here, of course, is that mode changes are infrequent relative to the experiments. Precise characterization of the relationship between mode switching behavior and window size also requires further research.

4 Case Study: Checking Wireless Sensor Network Applications

A wireless sensor network (WSN) usually comprises of a collection of tiny devices with built-in processors that gather physical information such as temperature, light, or sound, and communicate with one another over radio. WSN applications sit on top of an operating system called TinyOS [2]. TinyOS provides component-based architecture with tasks and event-based concurrency allowing applications to connect different components that coordinate concurrency via tasks and events. TinyOS has a small scheduler and many reusable system components such as timers, LEDs, and sensors. These components are either software modules or thin wrappers around hardware components.

TinyOS itself and WSN applications are written in nesC [6], an extension of C that provides a component-based programming paradigm. Before applications can be run on hardware, TinyOS itself and applications are compiled into C programs, which are then compiled again into specific hardware instructions. These hardward instructions can be downloaded directly onto the physical devices or a simulator. Most WSN applications are developed and tested on a simulator before they are deployed in the environment because on-chip testing and debugging are very difficult since it cannot tell a developer what causes the perceived errors. A simulator usually produces detailed execution steps taken in a program and allows a developer to examine his or her program to find bugs or errors.

However, the data returned by a simulator may be too detailed and overwhelming making the process of finding errors difficult. This case study takes a higher level approach by using MaC to aggregate the simulator data and allows developers to formally specify specific patterns of bugs or properties that an application must hold in an aggregate fashion. Properties of WSN applications may be specified to examine periodic behaviors, identify a faulty node, and analyze send and forward behaviors. MaC then monitors the application's data produced by a simulator and checks the data against the application's specification. In this case study, we use Avrora [14], a widely used simulator for WSN applications. Avrora provides an instrumentation capability for MaC to retrieve information about each sensor node running on the network environment within Avrora. It allows MaC to use this information to monitor and check applications run on Avrora against their specification requirements.

The result of the monitoring and checking allows us to gain some understanding of relevant behaviors of wireless sensor devices and can narrow the gap between the high-level requirement and the implementation of an application.

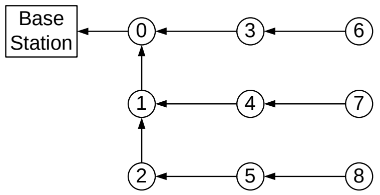


Fig. 3. Possible routes discovered by Surge

For the case study, we chose Surge application [6]. Surge periodically samples a sensor to obtain environment information such as light or temperature and reports its readings to a base station. Before sampling, each node discovers a multi-hop route to a base station in terms of a spanning tree by sending messages to its neighbor and then establishing an appropriate node as its *parent*. After the route is discovered, each node samples environment and sends data to its parent, which then forwards to its parent until the data arrives at the base station. When Surge is run on Avrora, node locations must be supplied to the simulator. In this paper, nine nodes are formed in a 3×3 grid. Figure 3 presents possible multi-hop routes that can be discovered by Surge. Surge consists of different TinyOS components such as a timer, a multi-hop router, and a sensor, among others. Surge wires these TinyOS components appropriately and implements operations such as a task `SendData` that reads a sensor and sends data. Tasks from Surge and TinyOS are logged and sent to MaC to be checked against Surge's specification.

4.1 Identifying a faulty node.

One property in the Surge application is to identify a faulty node using probabilistic properties. A node can be identified as faulty if it often fails to send data periodically or stops sending data where *often* means with probability of 0.15. It can be written in terms of a MaC alarm as $failSend\ pr(> 0.15, sendData)$. It states an alarm should be raised when a task misses its deadline with a probability > 0.15 given that an event *sendData* has occurred. Thus, the observed probability $\bar{p}$ can be calculated as follows.

$$\bar{p} = \frac{\mid failSend \ \&\& \ (sendData \ || \ end([sendData, failSend])) \mid}{\mid sendData \mid}$$

Once the properties are specified, MaC can check Surge via Avrora against these properties. The faulty node error is a physical error rather than a software error. Because the environment simulated by Avrora is perfect, it is impossible to detect this error on Avrora unless an artificial bug is introduced into Avrora to simulate the unpredictable environment of sensor nodes. In this paper, using the java class `Random`, an artificial bug is introduced into nodes 1, 4, and 8, shown in Figure 3. The faulty nodes would fail to send a message with probabilities indicated in Table 1. Given the window size of 80 and a 97.5% significance level (which yields $z_\alpha = 1.96$), an alarm is raised only for Node 4. Note that the alarm is not raised for Node 8, which barely exceeds the threshold. The absence of an alarm means that the difference is not statistically significant for the given confidence level.

Node	**Number of Sends**	**Number of Failss**	$\bar{p}$	z	**Alarm**
0,2,3,5,6,7	80	0	0.0	-3.757	No
1	**80**	**5**	**0.0625**	**-2.192**	**No**
4	**80**	**19**	**0.2375**	**2.192**	**Yes**
8	**80**	**13**	**0.1625**	**0.313**	**No**

Table 1. Probabilistic send in nodes 1, 4, and 8

5 Conclusions

Probabilistic property specifications for runtime verification are needed because system behaviors are often unpredictable due to ever-changing environments and cannot be checked statically. Unlike model checking, which can sample multiple execution paths, runtime verification operates on a single execution path, which needs to be decomposed into non-overlapping samples. We present a technique to decompose a trace into several samples based on specification of two kinds of events: one that starts an experiment and the other that denotes its successful completion. Once samples are collected, the probability of success is estimated, and the probabilistic property is checked using hypothesis testing. This paper extends our earlier work [12] by presenting a cleaner semantics based on conditional probabilities and discusses a new implementation approach that is based on a sliding window of samples considered in the hypothesis testing. We present case study, in which we monitor executions of wireless sensor network applications via a simulator. Future work includes a more thorough analysis of the sliding window technique and a more complete case study of the WSN application, directly via their physical devices instead of a simulator.

References

1. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 44–57, Vanice, Italy, 2004.
2. D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. In *Proceedings of the ACM Conference on Embedded Systems Software (EMSOFT)*, Tahoe City, California, October 2001.
3. J. L. Devore. *Probability and Statistics for Engineering and the Sciences.* Duxbury Thomson Learning, 2000.
4. D. Drusinsky. Monitoring temporal rules combined with Time Series. In *Proceedings of the 2003 Computer Aided Verification Conference (CAV)*, July 2003.
5. B. Finkbeiner, S. Sankaranarayanan, and H. B. Sipma. Collecting statistics about runtime executions. *Formal Methods in System Design*, 27(3):253–274, 2005.
6. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.
7. K. Havelund and G. Roşu. Java PathExplorer – A runtime verification tool. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2001.
8. J. Jayaputera, I. Poernomo, and H. Schmidt. Runtime verification of timing and probabilistic properties using WMI and .NET. In *Proceedings of the 30th EUROMICRO Conference*, 2004.
9. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a runtime assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, March 2004.
10. K. J. Kristoffersen, C. Pedersen, and H. R. Anderson. Runtime verification of Timed LTL using using disjunctive normalized equation systems. In *Proceedings of the 3rd International Workshop on Runtime Verification*, 2003.
11. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.
12. U. Sammapun, I. Lee, and O. Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In *Proceedings of the 11th IEEE International Conference of Embedded and Real-Time Computing Systems and Applications*, 2005.
13. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proceedings of the 5th International Workshop on Runtime Verification*, July 2005.
14. B. L. Titzer. Avrora: The AVR simulation and analysis framework. Master's thesis, University of California, Los Angeles, June 2004.
15. A. Wald. *Sequential Analysis.* Dover Phoenix Editions, 2004.
16. H. L. S. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking: An empirical study. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2004.