Journal of Theoretical and Applied Information Technology
15th February 2018. Vol.96. No 3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

CLASSIFICATION OF CONCURRENT ANOMALIES FOR IOT SOFTWARE BASED SUPPORT VECTOR MACHINE

[1]ZHIQIANG WU, [2]ASAD ABBAS, [3]XIN CHEN, [4]SCOTT UK-JIN LEE*

[1,2,3,4]Department of Computer Science and Engineering, Hanyang Univerisity, Ansan, South Korea

E-mail: {[1]wzq0515, [2]asadabbas, [3]xxtx0122, [4]scottlee}@hanyang.ac.kr

Corresponding Author: SCOTT UK-JIN LEE

ABSTRACT

Internet of Thing (IoT) can connect anyone with anything at any point in any place. Currently, growing number of IoT devices have become a major role of daily life owing to their convenience. The IoT devices usually controlled by Web applications and mobile applications, which will process lots of events from user's controller to devices. Hence, such software is a kind of concurrent program in IoT environment because the software is unable to simultaneously process these events, which may cause the concurrent issue. There is event-drive model in either Web application or mobile applications, which is unable to easily detect the concurrent anomaly by existing approaches due to the non-determined of execution and hardly reproduced by the same sequence. The previous techniques of concurrent detection are excessive limitations that only used for one of concurrent anomaly with the large number of false positive. In this paper, we describe a novel methodology to dynamically classify two types of concurrent anomalies for IoT software. According to the executable sequence graph, we generate the training and test examples for classification. The vectorization features are classified by Support Vector Machine (SVM) with Gaussian kernel. The SVM will predict the concurrent state of current executable example. As a result, the optimal true positive of simulation is 80% in our experiment which is a higher accuracy than others.

Keywords: *Concurrency Anomalies, Machine Learning, IoT Software, Support Vector Machine, Classification*

1. INTRODUCTION

The Internet of Things (IoT) is an emerging concept, which contains a large number of heterogeneous and pervasive things that steadily generate information about our real world [1, 2]. These devices connected to others in the same network such as objects, machines, vehicles, home appliance and other physical systems. However, these devices are various embedded systems, smart phone, tablets, etc. The Operation Systems which non-professional users can use have the Graphical User Interface (GUI) mobile systems such as WinCE, Android OS and iOS. IoT environment can be found everywhere in our daily life. The IoT applications have the ability of these ubiquitous and mobile connectivity such as WLAN, GPS, sensor inputs and Bluetooth.

Currently there are few kinds of IoT platforms for its software that is used to manage and control the IoT devices. For non-professional users, the mobile platforms are very convenient to manipulation. There are two most popular mobile platforms for IoT software, namely Android OS and iOS, which are both steadily upgraded and developed over time. Android OS is an open source, plays a major role as a platform for IoT software [3, 4].

Recently, the lower performance of IoT devices cannot be satisfied with user's requirements that prevents that users have a good experience especially for Android devices. The performance of Android OS will be decelerated by the over time. This issue has attracted many researchers to improvement of the performance of mobile systems. But, IoT software has received relatively less attention [5]. Current mobile systems which used for IoT environment are different from the traditional OS in computer. The mobile systems have the exclusive purposes in designated places for IoT. Hence, these systems have few characteristics, such as limited resources, event-based applications, and Linux-based kernel.

IoT software became universally existent in real world with convenient manipulation and services.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The IoT software is also a kind of concurrent issue due to the undetermined of execution from the huge of data stream in IoT environment, which may cause the collapse of applications. Nowadays, many kinds of methodologies are used to detected and solve concurrent anomalies, such as DEvA [6] and RacerDroid [7]. These methodologies only focus on one of concurrent anomaly in each approach. Furthermore, their approaches lead to the large number of False Positives (FP). And other researcher [8] attempted to prune the FP and improve the true positive of anomaly detection. IoT software provide an event based model for high concurrent software, which is different from the thread based mode. Mobile devices that are available for IoT platforms have a rich mixture of sensors and diverse ways of inputs that generate may asynchronicties to the data stream for IoT software. The event-based model integrates inputs from various sources, such as user input, touch screens, accelerators, etc. [8]. The existing detection tools of concurrent anomalies focused on thread-based programs. Even if these triggered events will be processed in a main thread, but there is logical concurrency among the most of events while these events may not be changed the order of processing by any developers.

The anomalies are difficult to recur by the same sequence of user's actions due to the undetermined schedules of potential events and limited system resources in IoT. Whereas the existing methodologies which are applied in concurrent systems for anomalies detection, focused on detecting one of concurrent anomalies that causes increased system overhead for concurrent anomaly. We propose a classification approach for two kinds of concurrent anomalies such as deadlock and data race [9]. These two anomalies are the most common issues in concurrent programs. The feature vector of deadlock and data race in IoT environments will be deduced in this paper. These executable sequence as features will be vectored to training samples for classification. The different sequences of executable programs will be classified by Support Vector Machine (SVM) with Gaussian kernel [10] in Octave [11] according to the features of executable sequences. We generated 22 samples to simulate our methodology. The result shows the prediction True Positive (TP) is 80% that is higher than previous works. Hence, the performance is improved by SVM model for IoT software. The concurrent anomalies can be efficiently exposed by this method with lower FP.

The rest of this paper is organized as follows. Section 2 describes the background of this study. The related work of this study is described in Section 3. Section 4 and Section 5 present the implementation of the concurrency anomaly classification using machine learning with SVM model and simulation result respectively. Finally, Section 6 concludes this paper and future work.

2. BACKGROUND

In this section, we briefly describe the related background of concurrent systems and IoT software.

2.1 IoT Software

IoT software are applied for portable and embedded devices that have become the most popular with rapidly increase in the amount and complexity of their capabilities. The mobile applications which are used for IoT environment replaced the role of traditional software for its portability and the convenient number of services. However, there are some differences between traditional software and IoT software, as described in Table 1 as below.

Table 1: Comparison between Mobile and Computers

Items	IoT Devices	Computers
Processors	RSIC	CISC
Computing	Lower	Higher
Input	Sensors, touch screen	Mouse, keyboard
OS Resources	Limited	Entire
Storage	Limited	Sufficient
Software	Event-driven	Thread-driven
Connectivity	Wi-Fi, 3G, 4G, LAN	LAN, Wi-Fi

The IoT software are easier to encounter the threat of concurrent issues owing to the limited resources and system architectures. Nevertheless, the event-drive program has three steps to interleaving execute, which are still processed in the main thread. When the steps handle the I/O operations, a callback will be registered to events cycle. After I/O operation, the prior step may continue its next procedures. The cycle of events accesses all events that are in the event queue when the triggered events will be allocated to a callback, which exists the non-determined schedules of events. The software may be crashed because the unknown event triggered with limited system overhead.

2.2 Concurrent Anomaly

The concurrent anomaly is a common situation in multithreading software, and these anomalies also

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

exists in event-driven programs because it belongs to concurrent systems. These anomalies are not only difficult to recur but also quite challenging to detect with the undetermined of concurrent manipulations. Fiedor, et al. [12] described their work on the uniform classification of concurrent anomalies. The five kinds of common anomalies in concurrent systems, namely, data races, atomicity violation, order violations, deadlocks and missed signals. We propose a classification for two most common anomalies of concurrent systems in IoT software, which are deadlock and data race.

2.2.1 Data Races

Data race is one of the common anomalies in concurrent systems. The two situations can distinguish data race with other anomalies in concurrent programs: a) where variables are shared with two or more events; b) where any given multiple accesses to a given shared variable are synchronized in some ways.

Therefore, a software contains data race if and only if it includes at least two non-synchronized accesses to a shared memory address where one of them is a write access [13]. The data races will be occurred while the write accesses have conflicts with others.

2.2.2 Deadlock

The other common anomaly is deadlock in concurrent systems. This anomaly exists in OS, distributed system and resources allocation system. Deadlock describes a status of events blocking due to the irrelevant invoking between events with the limited system resources. When system occurs this situation that cannot be recovered, these permanent blocking events or threads are called deadlock anomaly.

Deadlock occurs under four conditions: a) the threads/events shared multiple system resources, such as memory, CPU usage, etc.; b) the threads/events hold partial required resources and are waiting for others to continue the processing; c) the held resources cannot be released before the threads/ events completion; d) multiple events/threads exist in which each task holds other resources that are requested by the other task in the chain.

Thus, the deadlock anomaly will be triggered as a cycle while the multiple events mutually require the other resources. There is an example of deadlock is shown in Figure 1.

Dijkstra presented an methodology for deadlock avoidance named Banker's algorithm in 1965 [14], which is applied extensively to OS. Nowadays, the approaches of deadlock detection have various techniques with graph/tree algorithm to re-allocate the required resources reasonably, such as blocked-tree [15] and mutex tree [16], which try to detect a cycle in the execution sequence graph. For static techniques, the most common method is that the possible execution sequence in the program will be analyzed the order between events.

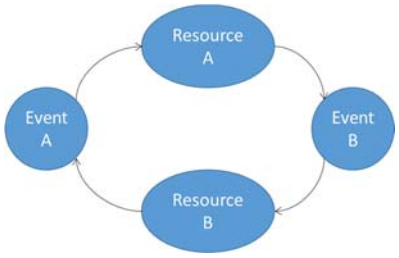

Figure 1: Example of Deadlock

2.3 Vector Clocks

Vector clocks [17] is an approach for generating the partial ordering of events in concurrent systems by F. Mattern in 1989. The execution relationship is presented by vector clocks between events in the systems. Moreover, each event in this executable sequence will be presented by a bit number for invoking relationship.

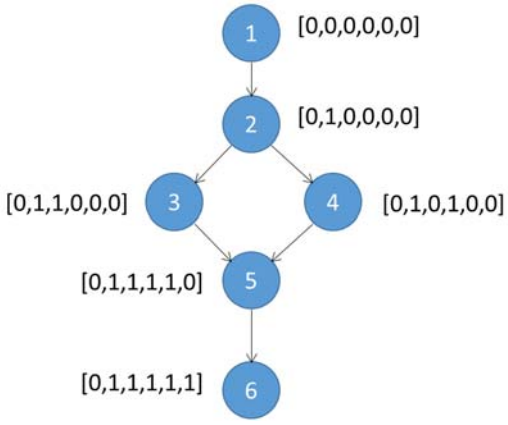

Figure 2. Vector Clocks with Bit Code

The vector clocks with bit code are applied in our methodology. The vector clocks of each node in the graph represents the compact use of the bit code as the weight of each fan-in event will always be incremented by the fan-in. the vector clock is easily used for vectorization of events for classification using SVM. An example of vector clocks with bit code in Figure 2 has the initialization vector that is

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[0, 0, 0, 0, 0, 0] because the root node is an original event without any fan-in. Therefore, those leaves should be incremented the values. The calculation of weight is formulated as:

$$VC_i(j) \coloneqq VC_i(j) + 1 \quad (1)$$

2.4 Event-driven Programs

We describe the event-driven programming model in this part. The event-driven programming model is used in the current mainstream mobile systems. The event-driven applications generally exist on mobile devices. The event-driven model is one of the common programming paradigm in software engineering. The executable paths are decided by triggered events, which could be generated by all events from external equipment to an application and internal thread in another application. The event-driven model has event cycle characteristics. The callback mechanism will be triggered in the related threads when external events occur. Moreover, two other paradigms exist, named synchronization (single-threaded) model and multi-threaded model. The three-programming paradigm models were compared at below.

Single-threaded Model: The tasks are sequentially executed according to the order of triggered tasks in the same thread for single-thread model. If a task is blocked due to I/O operations, all other tasks which are on waiting status, will not be successively executed until the previous task is finished.

Multi-threaded Model: The multiple tasks are separately executed in different threads, which managed by the OS. However, these threads should share one I/O operation. If the one thread holds the system resources, other threads that request the same system resources should be blocked until the resource released. This model is more effective than the single-threaded program, but the developers must write some related codes to protect the exclusive resources. Multi-threaded programs are difficult to curtail because they handle thread problems using synchronous mechanisms such as locks, critical section and thread local memory. If the execution cannot be correctly implemented within the program, it will generate subtle errors.

Event-driven Model: The multiple events are interleaving executed, which are still controlled in the main thread. When the events process the I/O operations, the callback should be registered to the event cycle. After the I/O operations, the previous event can continue its execution. The cycle of events accesses all events that are in the event queue when the triggered events will be assigned to a callback, which is waiting to process the events. A simple architecture of event-driven programming model is shown in Figure 3.

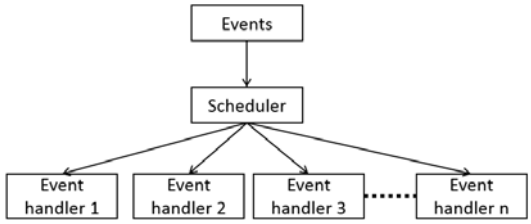

Figure 3. Architecture of Event-driven Programming Model

3. RELATED WORK

The concurrent detection is a popular research issue in concurrent and parallel systems. And these researchers made the various methodologies for different anomalies. We describe the recent research techniques on concurrent detection by diverse methods of analysis.

3.1 Static Analysis

Static analysis in IoT software is critical owing to the quality and reliability for software, which are vital keys to mobile application market in Android OS. The developers want to reach an anomaly-free software, but they will not know the exact executable sequence which will be triggered by users. Though developers can traverse the all possible executable sequences, and some condition still may omit because of the different environment of devices. The static analysis is very difficult to detect concurrent anomalies due to the non-deterministic execution of events. The static analyzers detect the software by reviewing the source code. All possible sequence of events could not be detected by this method.

Li, et al. [18] presented an approach to manifest concurrent anomalies in Android applications for generating the potential events by a hybrid-dynamic analysis. The AATT tool [18] in this paper is used to detect concurrent anomalies for Android applications. However, there are two limitations in this work. Firstly, the concurrent anomalies whose conflicting events cannot be manifested in different status. Secondly, the strategy for event scheduling is only re-scheduling events with a certain probability.

3.2 Dynamic Analysis

Dynamic analysis is a testing and evaluation technique for programs in rum-time. The objective is

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

to find anomalies in programs while the program is executed.

Tang, et al. [7] proposed a demonstrate called *RacerDroid*, which a lightweight scheduler by adapting the existed testing techniques. In this study, the race anomaly can be fully exposed by *RacerDroid* from the given potential data races, which can be generated from other existed detectors.

Glatz, et al. [16] developed a dynamic library for deadlock detection, which called *dpthread*. It builds up a resource allocation graph to detect the cycle in the executable graph using Depth First Search (DFS) algorithm. The evaluation shows a smaller performance and overhead in CPU usage and memory consumption without false positive or negative. However, this research is based on embedded Linux, which is different from event-driven programs.

Raychev, et al. [8] presented a dynamic race detection for event-driven programs using *EventRacer* in Web applications. The vector clocks applied to find the happen-before relation with chain decomposition. However, this approach used to web applications. There are some differences between web applications and mobile applications, such as sensor input and Bluetooth, which may cause harmful anomalies that cannot be detected by *EventRacer* in mobile applications.

3.3 Model Checking

Model checking is a method for formally verification of finite-state concurrent systems. It is an automatic technique for verifying distributed systems with concurrently executed processes. Some model checkers used in the current researched including CBMC (Bounded Model Chekcer for C/C++) [19], SPIN (Simple Promela Interpreter) [20] and JPF (Java-PathFinder) [21-23].

van der Merwe, et al. [23] proposed the JPF (Java-Path Finder) for Android, which is an Android application verification tool. This tool can successfully detect deadlock anomaly. The major challagen of JPF is to decide which part of the system should be modeled. If the most of Android framework are modelled, the more anomalies may be found in Android. However, scheduling prossiblities may exponentially increase when the more frameworks need to be modelled, which causes a huge work to veify. And, the JPF concentrates on verfiying a single application with several componets. Their work has a restriction that the parts of framework can be modelled and eliminated the possibility of finding more anomalies.

The existent techniques can expose many concurrent anomalies in concurrent systems. Those methodologies only focused on one of concurrent anomalies using a unique tool, which has a one to one relationship. However, these methodologies exist generality and precision issues in the results of detection.

Overall, the current methodologies still exist the large number of false positive and singularity. The previous research only focused on the one of concurrent anomalies to detect, which cause the OS should consume more system resources to execute multiple detection tools for various anomalies. However, the aim of the detection methodolodogies is used for improving the preformance of mobile OS. Thus, the presented method in this paper is able to detect the multiple concurrent anomalies in the same algorithms to reduce the system overhead. In this paper, we proposed a novel methodology to detect simultaneously multiple concurrent anomalies with a lower false positive in our result by SVM that has been applied in concurrent software field. And the methodology will be introduced in detail in next section.

4. METHODOLOGY

We descirbe the methodology that our proposed. We apply the Event language to generate the executable sequences of events in IoT software. The classification of concurrent anomalies consists of three main parts:

a) Generating executable sequence graph by source code and distinguishing the read and write operations.

b) Generating the vector clocks of events according to a directed graph that bulid on executable sequence graph, which will unify vectorization to a one-dimension vector for each executable path.

c) SVM with Gaussian kernel applied to classify three types of concurrent states which include data race, normal and deadlock.

Overall, the whole procedure of concurrent anomalies classification has five steps. Firstly, the example data should be input as the training data. Then, each example data will be hold a value that represents the concurrent status. The definited example data will be trained using SVM models before classification. The likelihood of model is able to process the features of data to predicit the categories for new event sequences. Finally, the test data can be input to the algorithm and get the prediction values of classification.

Journal of Theoretical and Applied Information Technology
15th February 2018. Vol.96. No 3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4.1 Executable Sequence Graph

We extract the source code from this application to generate the executable sequence from a IoT software in Figure 4.

This source code is a simple aircondition controller in IoT environment. There are only three functions in this application. Two buttons are to be clicked for changing the setting temperture such as up and down. Moreover, two buttons, which represent two events in this application, and the functions have respective *OnClickListener()* to change temperture by *changeTemperture()*. However, data race may ocurr in this simple application due to the non-certain executions.

```
1.  private int tempertture;
2.  private boolean flag;

3.  private void initView() {
4.    btn1 = (Button) findViewById(R.id.btn1);
5.    btn2 = (Button) findViewById(R.id.btn2);
6.
7.    btn1.setOnClickListener(new OnClickListener(){
8.      @Override
9.      public void onClick(View v) {
10.       flag = false;
11.       changeTemperture();
12.     }
13.   });
14.
15.   btn2.setOnClickListener(new OnClickListener(){
16.     @Override
17.     public void onClick(View v) {
18.       flage = ture;
19.       changeTemperture();
20.     }
21.   });
22. }

23. private void changeTemperture(){
24.     if (flag == false)
25.         temperature ++;
26.     else
27.         temperature -- ;
28. }
```

Figure 4. Part of Source Code in Controller

If the two buttons clicked with enough speed simutaneously, the evetns will be triggered in the short lag time. The two events will be sent to the event queue, and each event will be split to multiple tasks, which will be individually processed in the main thread. There is a looper thread for each application to process various kinds of tasks. All tasks can be executed in this thread, which causes a determinant executable sequence in concurrent systems. Thus, there exist the same issue in this IoT controller software owing to the unknown executions. For instance, if only one of buttons is clicked at a time, one event will be triggered and processed alone at the same time without any data race or resource limitation, which is a normal execution as a concurrent program. Meanwhile, the source code will generate a lot of situations when the two *OnClickListener()* are triggered simultaneously. There is an extreme possibility to ocurr the concurrent anomaly. If one of events is processed faster than other, these two events will be handled smoothly. Otherwise, the two events will be interleaving executed. If there is at least one writing instruction, the data race will be ocurred.

The write and read instructions should be distinguished by the source code to generate the executable sequence. Therefore, the executable graph represents the relationship of execution between events. This example in Figure 4 contains two shared variables *temperture* and *flag*, and a private method *changeTemperture()*. Figure 5 shows an executable graph for the aircondition controller using event semantic. The solid lines represent the happens-before relation, and the dotted lines represent the existence of some anomalies because at least one write instruction accesses the shared variable. Thus, the shared variables may be changed during the current processing event.

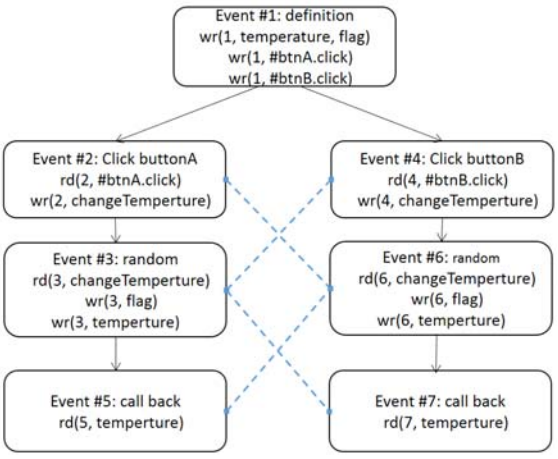

Figure 5. Event Semantic Graph for the Program in Figure 4

Figure 5 shows all executable paths that can be triggered by this concurrent program. Two buttons invoke the same method *changeTemperture()*. If these listener events are individually processed, this program will success change the temperture because

Journal of Theoretical and Applied Information Technology
15th February 2018. Vol.96. No 3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

there is no a confilict of write for shared variables. However, if these two buttons are triggered simultaneously, a special situation may be ocuured, such as the dotted lines in Figure 5. Assume that *button B* invokes the method *changeTemperture()*. Firstly, if the method only wrote a increment to the variable *temperture* before the callback to the listener event of *button B*, at this time, *button A* invokes the method *changeTemperture()* and change the value of *temperture*. At last, the value of *temperture* is not the original data when the program processes the latest read instruction in the listener event of *button A*. Thus, the data race ocurrs from this case as shown in Figure 6(c). Figure 6(a) shows all possible paths of executions for IoT controller application.

The executable sequence graph shows the sequences and relationships among asynchronized events. All possible executions represent by this graph and the sub-graph is a separate path of execution, which a individual executable sequence.

4.2 Vectorization

We manually created the executable sequence graph by this source code and expalined how data race ocurrs in the executable path. The feature of concurrent execution should be presented as one-dimension vector for classification fo concurrent anomalies using machine learning. Therefore, this graph converted to on-dimension vector by vector clocks.

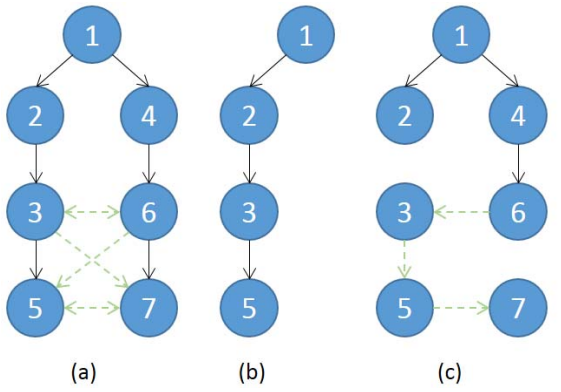

Figure 6. Executable Sequence Tree

Figure 6(a) is a directed graph, which consists of all executable paths of this controller application. However, all paths cannot be executed at the same time owing to the event-driven model. The main thread only handled an executabl path while an event is triggered. The Figure 6(a) can be divided to multiple executable paths, such as Figure 6(b) and Figure 6(c). According to the vector clocks of nodes, the matrix for features of executable paths is able to be generated for each path. For instance, the below matrix presents the status of one executable path in Figure 6(b). The vector clock of this node will be updated by formula (1) while the event executed the *i-the* node.

$$VT_b = \begin{matrix} 1 \\ 2 \\ 3 \\ 5 \end{matrix}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \quad (2)$$

The above matrix is a normal status of the executable sequence. The values of events in the current vector clock will be incremented while the program is going to the next event. The numbers which are out of the bracket are notations for each node. Therefore, *VT(5)* is the terminal vector clock. These vector clocks are combined in a matrix before vectorization of classification. The happens-before relations is able to be deteremined easily by the matrix. But this matrix still cannot be used for classification in SVM as a standard input. As a result, the feature vector which is used to machine learning as a one-dimension will be further extracted by this methodologies.

This 4 by 4 matrix presents an executable sequence. The multiple dimensional matrices is unable to be inputted as a feature vector for SVM model. The handled vector clock is a summation from the current vector minus the prior vector. Therefore, *n-1* (n represents the number of vector clocks) vectors will be generated by this method. The new summation of feature vector is able to get from these new vectors. The summation formula can be defined as:

$$V_{new} = \sum_{i=2}^{n} VT(i) - VT(i-1) \quad (3)$$
$$iff \;\; VT(i) \geq 0$$

This equation applied to calculate the sample graph for Figure 6(b). It is enable the simplification of feature vector for classification. The simplified vector clock as follows:

$$VT_{b\_feature} = [0\ 1\ 1\ 1] \quad (4)$$

This simplification vector by vectorization should be $VT_b(5)$, which is the latest event in the graph if this executable sequence is the non-anomalous state.

Figure 6(c) represents an anomalous of executable sequence that is data race due to the multiple writing instructions simultaneously. In the executable sequence, each node is able to traversed by Depth-Frist-Search (DFS) algorithm. The generated order of vector clocks for each node is different from the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

original order by this traversed sequence. However, above formula still adapts to the anomalous sequence. The new feature vector for Figure 6(c) is $VT_{c\_feature} = [0\ 0\ 1\ 1\ 1\ 1\ 1]$. The $VT_{c\_feature}$ is not equal to $VT_c(7)$ that is the ending of this event sequence. This executable sequence is an anomalous sequence as a sub-graph of Figure 6(a).

$$VT_c = \begin{matrix}1\\2\\4\\6\\3\\5\\7\end{matrix}\begin{bmatrix}0&0&0&0&0&0&0\\0&1&0&0&0&0&0\\0&0&0&1&0&0&0\\0&0&0&1&0&1&0\\0&0&1&1&0&1&0\\0&0&1&1&1&1&0\\0&0&1&1&1&1&1\end{bmatrix} \quad (5)$$

In the same way, the vector clocks can be generated easily for Figure 7 by the executable sequence graph. Then, the feature vector for this graph is $VT_{deadlock} = [0\ 1\ 1\ 2\ 2]$. The $VT_{deadlock}$ is not equal to $VT_{deadlock}(2)$, which causes the executable sequence to generate a cycle of events in the graph.

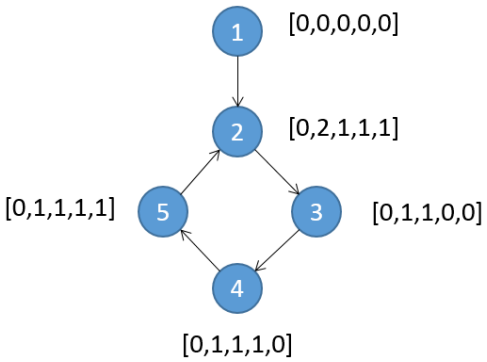

Figure 7. Executable Sequence Graph of Deadlock Example

4.3 Classification

In this part, we describe the classification method using SVM with Gaussian kernel. In our approach, the vector clocks denote the paths of executable events, and are used to classify the categories for concurrent anomalies. There are two steps for classification with SVM model.

4.3.1 Pre-processing

The thousands of events continously triggered at a time. The feature vectors are probably exceeding our predicted length, and they may have the different length among vectors. The generated feature vectors in prior section should be pre-processed as input for SVM before classification.

The feature vectors whose lengths are less than the initial length will be provided additionally as element 0 at the end of vectors for avoiding the jagged length. A horizontal vector that denote an executable sequence of events in IoT software will be inputted to SVM model for classification. Machine learning automatically captures the features between vectors by the training of these executable sequences.

Here, we generate 20 training examples by the source code from IoT software. Each node of vector represents one feature, and each training example has a notation y as its substantial concurrent state. For example, y represents a normal state if y equals 1. Similarly, it represents data race when y equals 2; the current executable sequence is a deadlock status if y equals 3. The complex examples which have a large of events are sorted to the unified format using the pre-processing.

4.3.2 Classification

This methodology is trying to increase the training TP and reduce the FP. The SVM with Gaussian kernel applied in this work to find out the best performance. The SVM hypothesis is defined as follows:

$$\min_\theta C\sum_{i=1}^{m}[y^{(i)}\cos t_1(\theta^T x^{(i)}) + (1-y^{(i)})\cos t_0(\theta^T x^{(i)})] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

$$\cos t_1(\theta^T x^{(i)}) = -\log\frac{1}{1+e^{-(\theta^T x^{(i)})}}$$

$$\cos t_0(\theta^T x^{(i)}) = -\log(1-\frac{1}{1+e^{-(\theta^T x^{(i)})}}) \quad (6)$$

where m is the number of training samples.

Moreover, the Gaussian kernel is used to fit θ. We should choose the values of C and σ carefully to reach a higher TP with the lower FP. The different combinations of C and σ will show the diverse performance of prediction which shown in Section 5.

This methodology will classify three classes of concurrent states in IoT software, which belongs to multip classes classification in SVM with multiple features. Nevertheless, the one to one method of SVM fro multi-class is used in this work. The three pairwise of classes are created as input data by the formula in vectorization section.each training set will be detected respectively in the simulation. When an example predicted by three SVM models, this example will get three results of predition by each combination of classes. SVM model will output a value to present a concurrent state that is the predicted result for the example. The number of predicted times for each class is able to calculate the result. The maximum number in the result of concurrent states is the final predicted result;

otherwise, it will be re-classified until get a specific result. The calculation model is shown in Table 2 which used a normal example.

In the next section, we will show the experiment result in detail and analyze the performance of this method.

Table 2. SVM Models

SVM Model	Normal	Deadlock	Data Race
(Normal, Deadlock)	1	0	/
(Normal, Data Race)	1	/	0
(Deadlock, Data Race)	/	0	1
Sum	2	0	1

5. IMPLEMENTATION RESULT

In this section, we describe the performance for our implementation by simulation. We generated 15 training examples and artificially distinguish their concurrent states by the source code inclu

ding an example in Figure 4. The 20 testing examples include the training exmaples which also will be detected by machine learning to calculate the precision. And there are nine examples of the non-anomalous, five examples of data race and six examples of deadlock.

The featire vectors that generated in vectorization will be applied in this section. There are two parameters for input which include a feature vector of executable sequence and an identifier of concurrent states. We should classify three types of status that mentioned in previous part. Then, the three group SVM models will predict each executable sequence.

Table 3. Simulation Results when C=1

σ	0.01	0.03	0.1	0.3
TP	70%	75%	75%	75%
σ	1	3	10	30
TP	80%	65%	70%	70%

The ideal performance is able to find out by adjusting the values of C and σ in SVM with Gaussian kernel. The C is the tolerance of fault. The tolerance is the lowest, which easily cause the overfit while C is too large; otherwise, the performance will be the lowest in the implementation. The σ denotes the number of training examples, which can effective the number of support vector and the speed of prediction.

Therefore, we applied the control variable method to find out the optimal combination of two related parameters. As a result, when σ is constantly increased, it causes the higher basis, and the TP is increased until σ equals 1 as shown in Table 3. On the contrary, the FP is decresed until σ over 1; then, FP will be increaed to 30%, which leads to a higher variance.

According to this simulation, the partial figure show the performance under the different values of σ

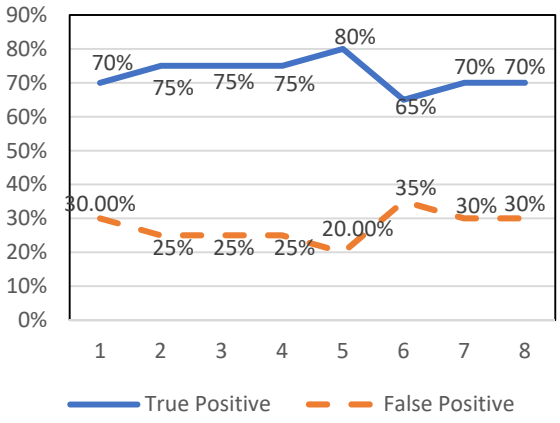

Figure 8. Simulation Result when C=1

when C equals to 1 as shown in Figure 8. The TP achieved the highest performance. When both parameters σ and C are equal to 1, the performance will reach the optimum condition. Nevertheless, the highest TP is 80% with the 20% false positive in our methodoloy. This performance still exceeds *EventRacer* [8]. This method is a novel and effective for classifying dynamically the concurrent anomalies by vector clocks for IoT software.

6. CONCULSION

The concurrent anomaly in IoT software is a popular issue. These current methodologies have the large number of FP and simplex concurrent anomaly without extensibility.

In this paper, we proposed a novel methodology for multiple kinds of concurrent anomaly in IoT software. This classification approach is able to be applied in IoT plarforms to detect the concurrent anomalies using machine laerning which is enable to predict the executable sequence. The machine learning used in this work to classify two kinds

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ofconcurrent anomalies. By the experimental simulation, the TP reached 80%, and FP decreased to 20% when the parameters *C* and σ equal to 1. Meanwhile, our method can classify the multiple anomalies in IoT software, that is an advantage with others.

Nowaday, seriously depend on various smart devices to either manipulate family appliances or monitor the real-time data. When a lot of requests send to the processing termination, the IoT environment is an individual concurrent system. Hence, a fluent operation in IoT software is essential.

We merely applied very limited examples to simulate the methodology. In the future, we will extend this method to automatically extract the executable sequence and vector clocks from the executing software. Meanwhile, this method will access to some open IoT software to increase the reliability and precision of simulation.

ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea through the Korean Government (MSIP) under Grant NRF-2016R1C1B2008624.

REFRENCES:

[1] S. Tata, R. Jain, H. Ludwig, and S. Gopisetty, "Living in the Cloud or on the Edge: Opportunities and Challenges of IOT Application Architecture," in Services Computing (SCC), 2017 IEEE International Conference on, 2017, pp. 220-224.

[2] I. F. Siddiqui, S. U. J. Lee, A. Abbas, and A. K. Bashir, "Optimizing lifespan and energy resources of smart meter in a green cloud-based smart grid," IEEE Access, 2017, pp. 20934-20945.

[3] É. Payet and F. Spoto, "Static analysis of Android programs," Information and Software Technology, vol. 54, no. 11, 2012, pp. 1192-1201.

[4] A. Abbas, I. F. Siddiqui, S. U. J. Lee, and A. K. Bashir, "Binary Pattern for Nested Cardinality Constraints for Software Product Line of IoT-Based Feature Models," IEEE Access, vol. 5, 2017, pp. 3971-3980.

[5] Q. Wang et al., "Multimedia IoT systems and applications," in Global Internet of Things Summit (GIoTS), 2017, pp. 1-6.

[6] G. Safi, A. Shahbazian, W. G. Halfond, and N. Medvidovic, "Detecting event anomalies in event-based systems," in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 25-37.

[7] H. Tang, G. Wu, J. Wei, and H. Zhong, "Generating test cases to expose concurrency bugs in android applications," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 648-653.

[8] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in ACM SIGPLAN Notices, vol. 48, no. 10, 2013, pp. 151-166.

[9] Z. Wu, "Support Vector Machine Based Concurrency Anomaly Classification for Mobile Applications," Master of Science, Hanyang University, 2017.

[10] S. S. Keerthi and C.-J. Lin, "Asymptotic behaviors of support vector machines with Gaussian kernel," Neural computation, vol. 15, no. 7, 2003, pp. 1667-1689.

[11] S. Sonnenburg et al., "The SHOGUN machine learning toolbox," Journal of Machine Learning Research, vol. 11, 2010, pp. 1799-1802.

[12] J. Fiedor, B. Křena, Z. Letko, and T. Vojnar, "A uniform classification of common concurrency errors," Computer Aided Systems Theory–EUROCAST 2011, pp. 519-526.

[13] W. Mansky, Y. Peng, S. Zdancewic, and J. Devietti, "Verifying dynamic race detection," in CPP, 2017, pp. 151-163.

[14] A. Tanenbaum, "Modern operating systems", Pearson Education, Inc., 2009.

[15] T. Shimomura and K. Ikeda, "Waiting blocked-tree type deadlock detection," in Science and Information Conference (SAI), 2013, pp. 45-50.

[16] B. Glatz, R. Beneder, M. Horauer, and T. Rauscher, "Deadlock detection runtime service for Embedded Linux," in IEEE Conference on Emerging Technologies & Factory Automation , 2015, pp. 1-7.

[17] F. Mattern, "Virtual time and global states of distributed systems," Parallel and Distributed Algorithms, vol. 1, no. 23, 1989, pp. 215-226.

[18] Q. Li et al., "Effectively Manifesting Concurrency Bugs in Android Apps," in Software Engineering Conference (APSEC), 23rd Asia-Pacific, 2016, pp. 209-216.

[19] S. Falke, F. Merz, and C. Sinz, "The bounded model checker LLBMC," in IEEE/ACM

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

International Conference on Automated Software Engineering (ASE), 2013, pp. 706-709.

[20] A. Maleki, "Framework for Analyzing Highly Concurrent Algorithms In SPIN," 2011.

[21] S. Iqbal, S. U. Shah, M. Nauman, and M. Amin, "Extending Java Pathfinder (JPF) with property classes for verification of Android permission extension framework," in IEEE International Conference on System Engineering and Technology, 2013, pp. 15-20.

[22] A. Kohan et al., "Java Pathfinder on Android Devices," ACM Software Engineering Notes, vol. 41, no. 6, 2017, pp. 1-5.

[23] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using Java PathFinder," ACM Software Engineering Notes, vol. 37, no. 6, 2012, pp. 1-5.