Towards Model Checking of Network Applications for IoT System Development

Hing Ratana and Sharifah Mashita Syed Mohamad
School of Computer Sciences, Universiti Sains Malaysia, 11800 Gelugor, Pulau Pinang, Malaysia.
ratanahing@student.usm.my

***Abstract*—With the expansion of the Internet, Internet of Things (IoT) gains lots of interest from industries and academia. IoT applications enable human-to-device and device-to-device interactions. For a successful deployment of IoT systems and services, software reliability is a very important requirement for IoT to ensure that data/messages have been received and performed properly in a timely manner. The concurrent connections of embedded sensors and actuators are non-deterministic in nature which makes testing insufficient to guarantee program correctness. In contrast, model checking techniques explore the entire behavior of a system under test (SUT) in brute-force and systematic manner. It investigates each reachable state for different thread schedules. Recent model checking techniques have been applied directly to networked programs. This paper reviews model checking techniques for networked applications and presents their strengths and limitations. A preliminary proposal for model checking of networked applications that addresses the identified gap is presented.**

***Index Terms*—Cache-Based Approach; Internet of Things Applications; Network Model Checking; Software Reliability.**

I. INTRODUCTION

With the expansion of the Internet, the Internet of Things (IoT) gains lots of interest from industries and academia. IoT projections suggest that, by 2020, there will be 50 billion connected devices and $19 trillion opportunity into the IoT industry [1]. IoT represents a worldwide network of uniquely addressable inter-connected smart objects such as sensing and actuating devices that provide ability to share information across multiple platforms in order to enable innovative applications [2].

According to Lee and Lee [3], there are five essential IoT technologies for deployment of successful IoT-based products and services. These technologies are radio frequency identification (RFID), wireless sensor networks (WSN), middleware, cloud computing, and IoT application software. All these technologies involve with hardware and software communicating each other via network and the Internet. For instance, the middleware allows the mobile devices to perform communication and input/output with sensors and actuators. It hides details of different technologies of those smart objects.

Figure 1 illustrates typical IoT architecture using middleware that hides details of different technologies implemented by smart objects. The Internet provides uniquely addressable inter-connected points to the objects, and it is the central point for communication. The communication protocols such as TCP/IP, UDP and HTTP are used for communication amongst mobile devices, the Internet, middleware, and smart controller. Other protocols like 6LoWPAN, Z-Wave, and ZigBee used for communicating between the smart controller and smart objects. Smart homes for instance, light bulbs, light switches, water heaters, solar panels, motion sensors, window/door sensors can be programmed to connect with each other to the smart controller and from the smart controller to the Internet so that they can share information and assist home users in undertaking operational tasks such as turning off the lights in a house from a smart phone.

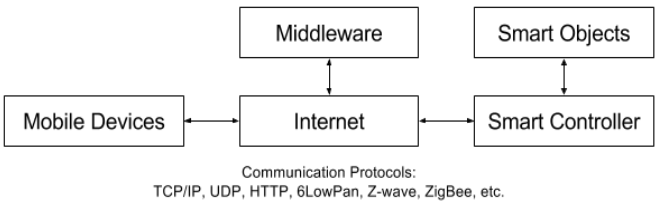

Figure 1: Typical IoT architecture via middleware

The reliability of IoT applications is one of the concerns for successful IoT deployment as described in [3]. The applications enable human-to-device and device-to-device interactions, and they need to ensure that data/messages have been received and performed properly in a timely manner.

Software testing [4] is one of the common practices to ensure the reliability of IoT applications, however, is depended on systematic guess and how well the software testers know about the system under test (SUT). The concurrent connections of wireless embedded sensors and actuators to the middleware are non-deterministic in nature. The interleaving between threads increases the challenges to software testers. In addition, setting up an environment and instrumenting the SUT are time consuming and expensive.

Model checking [4, 5], on the other hand, is one of the formal verification methods for ensuring the reliability of critical software system. It explores the entire behavior of a SUT in brute-force and systematic manner and investigating each reachable state for different thread schedules. Recent model checking techniques [6-8] have been applied to network programs. The goal of this work is therefore to review model checking techniques for networked applications and how these techniques can help with the development of reliable applications for IoT ecosystems.

This article is organized as follows: Section II provides the fundamental background to model checking techniques and the model checker tool; Section III describes the problems of model checking network applications and review its current works; Section IV proposes initial work toward verification of IoT application. Finally, the conclusion of the article is provided in Section V.

II. BACKGROUND

Software testing depends on systematic guess and how well the software testers know about the SUT. The concurrent connections of wireless embedded sensors and actuators to the middleware are non-deterministic in nature. The interleaving between threads increases the challenges to software testers. In addition, setting up an environment and instrumenting the SUT are time consuming and expensive. Software model checking, on the other hand, is one of the formal verification techniques, which is used to verify software system. This technique conducts an exhaustive search of all possible system states and, if encountered an error, it provides "counterexample" which tells us where the root of the bug is. The "counterexample" is the faulty trace that provides the important clues for fixing the problem.

In this section, we introduce the concept of software model checking and the model checker tool called Java PathFinder (JPF) and its extension for verifying network applications.

A. Model Checking

Model checking [4, 5] is one of the formal verification techniques that exhaustively checks for property violations in concurrent system. It explores all possible system states in brute-force and systematic manner. There are two major advantages of model checking over the other formal verification techniques. First, it is fully automatic. This means that model checking does not require any user supervision to control the input during the design simulation. Second, it provides "counterexample" when the given model does not satisfy the given properties. Counterexample is like a bug trace, which is import clues to fix the problem.

According to [9], the model checking problem can be stated as below:

$$M, s \models f \quad (1)$$

where M is a Kripke structure (i.e., state-transition graph) and f is a formula of temporal logic (i.e., the specification). The problem is to find all states s of M such that $M, s \models f$.

The system model is formally described as Kripke structure or transition system (TS), and the system properties are generally expressed in temporal logic. When the state of TS satisfies with the property, the model checking continues to the next state until the error is found. It proceeds until the end state. If the error is found, it produces the *counterexample* that gives important clue to fix the error. Model checking explores the entire state-space of the concurrent systems. The basic search algorithms are depth-first-search (DFS) and breath-first-search (BFS). These two search algorithms involve backtracking the state inside the programs. The example of backtracking is explained in section *B* later in the article.

Modern model checkers [10-18] have been applied directly to the actual implementation of the programs, written in standard programming languages such as C or Java. These tools help programmers to detect errors during the implementation phase. An example of model checker tool that model check real programs is Java PathFinder [14], which we will focus on in the rest of the article.

B. Java PathFinder

Java PathFinder or JPF [14] is a verification and testing environment tool for Java. It is an explicit-state model checker that verifies Java programs for concurrency defects, runtime analysis, and generation of test cases depending on how the user configures the verification properties as input. By default, JPF can check for deadlocks, race conditions, and unhandled exceptions (including Java *assert* expression). The tool is developed by NASA Aimes Research Center and became an open-source project in 2005.

Figure 2 illustrates the overall architecture of JPF. The tool requires the Java bytecodes (*.class) and its requirements (*.jpf) as its inputs, and it produces a report of the verification result as an output.

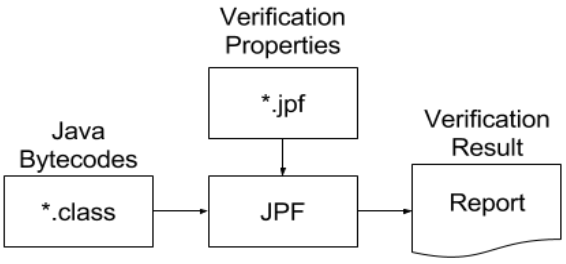

Figure: 2. Java PathFinder architecture

JPF is written in Java so it is executed on top of Java Virtual Machine or we call it host JVM, and the Java program under test is executed on top of JPF, which is running a customized JVM for model checking purpose or we call it as JPF_{jvm}. The different is that JPF_{jvm} involves backtracking; while host JVM does not involve backtracking.

As mentioned earlier, model checker requires backtracking the SUT. To illustrate this, let's look at the example of how JPF_{jvm} and host JVM executes the program as shown in Figure 3. Figure 3 is an example Java program that computes the two random variables *a* and *b*. The program starts with the initialization of Random class with value of 42. The integer variable *a* and *b* are initialized and given ".nextInt()" method with value of 2 and 3, respectively. Variable *c* does computation as shown in line 9. Finally, the program prints out the result of *c*.

```
1  import java.util.Random;
2  public class Rand {
3      public static void main (String[] args) {
4          Random random = new Random(42);
5          int a = random.nextInt(2);
6          System.out.println("a=" + a);
7          int b = random.nextInt(3);
8          System.out.println("b=" + b);
9          int c = a/(b+a -2);
10         System.out.println("c=" + c);
11     }
12 }
```

Figure 3:. Simple Java program using random class

Figure 4 (a) indicates the execution graph on host JVM, and Figure 4 (b) shows the execution graph of the program on JPFjvm. The octagon, single circle, and double circle represents the start state, reachable state, and end-state, respectively. Notice that in (a) the program executes on host-JVM. It does not involve backtracking thus the program does not cause any error. However, in (b), the JPF executes the program in all possible ways until it finds the error state.

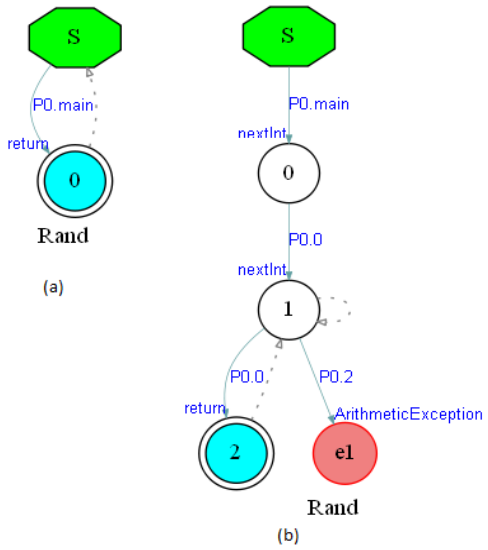

Figure 4: Execution graph by (a) host-JVM and (b) JPF_{jvm}

There are two main distinct strategies of JPF tool, the jpf-core and the extensibility of the jpf-core, that make JPF tool becomes the most useful model checker for finding bugs in concurrent Java programs.

The jpf-core consists of two main components such as the custom Virtual Machine or (JPFjvm) and the search. Jpf-core is a customized Java Virtual Machine (JPFjvm). It is a JVM which mean it allows Java program to execute, but this JPFjvm executes program differently from the host-JVM. The host-JVM executes only one path of the program at a time; while JPFjvm explores all the possible reachable system states of the program.

First, the search (default is depth-first-search) component inside the jpf-core represents the program states as a directed graph where the nodes represent its states and edges denote transitions (or choices). The choice generator creates the next successor state of the current state, and the search goes through the state one by one in a non-deterministic manner. If the current state does not have any successor state, the search backtracks to the previous visited state and explores the next unvisited states. By following this policy, the JPF theoretically explores all the possible system behavior of the program. The jpf-core produces a report that leads to the bug if there is a bug found inside the concurrent program.

Second, JPF allows the extensibility of its core to tackle different model checking problems. The jpf-core provides listeners, little "plugin", that let the user closely monitor all the actions by JPF such as executing single instructions, creating objects, reaching a new program state, and many more. Listeners are the most important extension mechanism of JPF. They provide a way to observe, interact with, and extend JPF execution with classes. Since listeners are dynamically configured at runtime, they do not require any modification to the jpf-core. Listeners are executed at the same level as JPF, so there is hardly any limit of what the user can do with them.

Finally, JPF is not able to backtrack native code such as system calls, input/output (I/O) that effects the host environment, accessing database, and network. For such cases, JPF provides model classes that simulate the native codes. All model classes must be developed and written in Java. Model classes can call native peer classes that run on the host-JVM to execute native methods such as network I/O.

III. Model Checking Networked Applications

Model checking explores the entire behavior of a SUT in brute-force and systematic manner and investigating each reachable state for different thread schedules. The SUT involves backtracking by the model checker tool. The problem happens when model check network programs. The SUT may repeat sending messages (I/O operations) to the external processes, however, the external processes, which are not under control of the model checker, cannot synchronize with the backtracking of SUT; therefore, the synchronization causes the direct communication between the SUT and external processes to fail.

The current approaches such as cache-based [19-22] and centralization [7], [23-27] have been applied to model check network programs. Below sections describe details and current works of cache-based and centralization, respectively.

A. Cache-Based Approach

The concept of cache-based [19-22] approach is to model check a single process inside the model checker and runs all the other processes externally in their native environment. A process is a self-contained execution environment and has their own resources such as memory, CPU time, and I/O devices, whereas threads run within a process and share the process runtime resources. In cache-based, the SUT and "peers" denote the single process inside model checker and the external processes, respectively. The SUT executed by the model checker is subjected to backtracking, while external processes run normally.

The main challenge of this approach is the synchronization between the SUT and its peers since the SUT is subjected to backtracking by the model checker, and the model checker does not have any control of its peers. During model checking SUT, the SUT may resend data which might interrupt the correct behavior of the peers, and the peers may not send the correct data back to the SUT. A special cache layer has been developed to solve these problems. Existing cache-based techniques [19-22] address this problem by introducing a special cache layer between the SUT and its peers for state synchronization.

Figure 5 illustrates the overall architecture for cache-based approach for model checking network applications. The model checker executes the SUT in exhaustive ways making the repeated requests. The special cache layer intercepts all the communications between the SUT and its peers. It represents the state of communication at different points in time. After the SUT backtrack, the data previously received by the SUT is responded by the cache when requested again. If the SUT resend the same data that previously in the cache, the data is not sent again over the network; instead, the data is compared to the data in the cache storage.

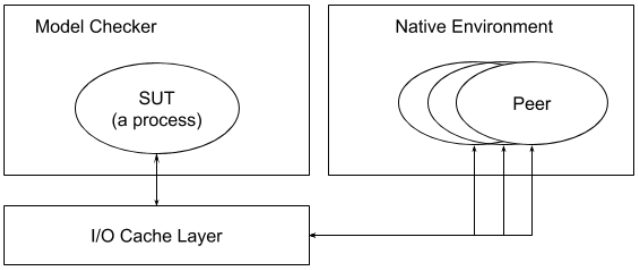

Figure 5: Overal architecture of cache-based approach

Initial work by Artho et al. [19, 20] proposes a solution for model checking network applications by developing a special caching layer for steam-based input/output (I/O). They introduce the idea of I/O caching via deterministic communication. We can refer it as *linear-time* cache. The solution works if the I/O operations of the SUT always produce the same data stream regardless of the non-deterministic of the thread schedules. The communication between the SUT and its environment must be independence of the thread scheduling. For instance, the client sends a sequence of characters to the server, the server supposes to send the same sequence of the characters back to the client, regardless of the thread schedules. If it is not the case, the behavior of the communication resource would be undefined.

The later work extends the idea of caching I/O communication traces to a wider range of applications by developing communication model that allows diverging communication traces between different schedules [21]. This concept is called *branching-time* cache. It allows for non-deterministic caching traces between the SUT and the peers, but it does not allow non-determinism within the peers. For this approach, the SUT at least can send different data from the previous observed ones.

To allow the non-determinism in peers, the proposed work in [22] combines a cache layer with process checkpointing [28]. Process checkpointing environment can run, pause, replay the peers at any point in time. During model checking of SUT, checkpointing idea can be incorporated when the SUT requires the synchronization of data from the peers, at that points, checkpointing can play and replay the peers' states accordingly to the requests from the SUT. By doing this, this concept gives a broader range of model checking network applications.

Cache-based techniques have been implemented into JPF extension called "net-iocache" [6]. This extension executes on top of *jpf-core*, and extra codes have been developed to control the behavior of the processes; for example, the control codes of executing sever process first before starting the client process. Sebih et. al [8] later extends net-iocache to verify network applications based on UDP protocols. The initial of their proposed work assumes that the communication packets can be lost, duplicated, and reordered. It is challenging to verify UDP-based applications due the unreliable connection; therefore, the authors simulate the behavior of the applications according to their requirements. However, they have added UDP support to net-iocache and successfully simulated UDP's unreliability by systematically generating combinations of packet lost, duplication, and reordering. Table 1 provides the summary of cache-based approach, its strengths and weaknesses, as well as the types of protocol supported for model checking.

Table 1
Model checking networked applications based on cache techniques

Cache-Based Approach				
Techniques	Protocol	Tool	Strength	Weakness
Linear-time cache [20]	TCP/IP	net-iocache	• Covers all I/O operations. • Complete execution semantics for steam.	• Non-deterministic of messages. • Very strict requirement.
Branching-time cache [21]	TCP/IP	net-iocache	• Covers more broader range of applications. • Allows SUT to send and accept different requests.	• Still deterministic peers.
Cache with process checkpointing [22]	TCP/IP	net-iocache	• Covers more broader range of applications. • Allows non-deterministic peers.	• Involves play and replay peers so the response may error prune.
Extension of cache-based for UDP [8]	UDP	net-iocache	• Covers modeling class of DatagramSocket in Java	• Simulating UDP behavior which is not so practical in real applications.

B. *Centralization Approach*

The concept of centralization techniques is to model check all processes within a model checker [7], [23]–[27]. These techniques can be applied at SUT level, OS level, and model checker level. Figure 6 shows the overall architecture of applied centralization techniques: (a) SUT level; (b) OS level; (c) Model checker level. At the SUT level, the processes are transformed into one main process. So, each process is mapped into a thread, and the model checker verifies the main process. Whereas the centralization can be applied at OS level. This technique does not involve transforming the SUT, instead, all the processes are running on top of virtualization tool, and model checker tool is extended to capture the state of the virtualization tool for state-space exploration. Finally, the model checker level aims to develop the model checker that can capture multiple processes within the tool itself.

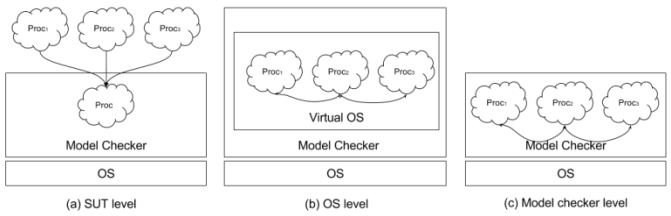

Figure 6 Overall architecture of applied centralization approach

1) *Centralization at SUT level*

The centralization approach can be applied at the SUT level. To achieve this, the processes are transformed into one main process; therefore, each process is mapped into a thread, and the model checker verifies the main process.

The main challenges of SUT level centralization are: how the techniques can map processes into threads, how to represent communications between transformed processes (local threads), how the start and end process semantics, and how to separate static functions and types between local threads. The related works that have been proposed to address some of the above issues are discussed below.

Initial work from Stoller and Liu [23] applied centralization technique at SUT level. They propose the concept of

transforming processes into a single process by replacing remote method invocation (RMIs) with the local ones that simulate RMIs. In addition, Stoller and Liu develop CentralizedThread class that extends Thread and initialize an instance of field type integer to denote process id. By doing this, they can map each process into a thread, and each thread communicate with each other via the simulated local RMIs.

Later work from Artho and Garoche [24] provides a more accurate transformation of processes into a single process, and they also address some of the limitations of previous work by Stoller and Liu. In contrast to previous work, Artho and Garoche perform bytecode instrumentation which is applicable to systems compatible with newer version of Java and, in addition, their technique is also applicable to applications that use sockets for communication.

Ma et. al [27] also propose technique for SUT level centralization. Their work extends the work from Artho and Garoche and addresses some of its limitations. They describe the techniques of handling class confliction by renaming classes that have identical names but have different bytecode. Ma et. al approach presents a way to terminate all processes by killing all their related threads.

Finally, the SUT level centralization has been proposed by Barlas and Bultan [25]. They are mainly focusing on environment generation by introducing a framework called Netstub. The Netstub API requires users to manually develop on how the environment should be generated to accommodate the SUT during model checking. In addition, Netstub also allows model checking a process at a time. The Netstub environment can generate network events which are perceived by the SUT.

2) Centralization at OS level

In centralization at the OS level, the processes are running on a virtualization tool; therefore, this approach does not require transforming the SUT. This approach requires the extension of model checker's scope to capture the state of the virtualization tool.

The major challenges for this technique is the state space explosion. Since the SUT processes are running on top of virtualization tool and model checker must cover all the processes including virtualization tool processes, this will lead to exponential growth of states.

Nekagawa et al. [26] develop a model checking framework based on this approach. Their proposed framework can execute very close to the actual model checking execution environment. They combine the user-mode Linux and the GNU debugger (GDB) to save and restore the entire Linux state. GBD can support several programming languages including Java. Processes are running on virtualization tool and once non-determinism is detected within a process, the state of the OS and any possible execution paths are computed and explored by the tool.

3) Centralization at Model checker level

Recent centralization approach has been implemented at model checker level by initial work from Shafiei and Mehlitz [7]. They develop multi-process JVM for JPF which allows model checking of distributed Java applications. To address the problems of class confliction, static functions and static fields, the new multi-process in JPF modifies the class loaders in JPF. The processes are mapped as a group of threads. During the initialization, each new thread is created by the SUT automatically. To capture scheduling points inside JPF, the new communication models have been developed based on network API calls. This technique has been implemented into JPF extension called "jpf-nas".

The major challenges with centralization at model checker level are managing the state-space within model checker, modeling internal communication between local threads, and possible covering of language API and classes. Table 2 summarizes the centralization approach, and some of their strengths and weaknesses.

Table 2
Model checking networked applications based on centralization techniques

Centralization Approach				
Techniques	Protocol	Tool	Strength	Weakness
SUT Level [23]–[25], [27]	TCP/IP RMI	Expect to integrate with Bandera, JPF	• Covers all errors including network states. • Does not require SUT transformation. • Covers some errors in the processes.	• Involves SUT transformation • Separate static functions and fields
OS Level [26]	TCP/IP	SBUML and GDB		• Exponential state-space explosion
Model checker level [7]	TCP/IP	jpf-nas	• Automatic load network applications and verification. • Does not involve SUT transformation.	• Does not support UDP. • Managing state-space and communication within the tool.

C. Summary

The cache-based techniques verify one process at a time, while letting the rest of processes run externally in their native implementation. The SUT is subjected to backtracking that brings the challenges of state synchronization between the SUT and its peer processes. First, Linear-time cache can handle network applications that the request and response have the same sequence of data regardless the thread schedules, if not otherwise, the behavior of the communication will fail. Second, branching-time cache can let the SUT send different messages to their peers. However, it does not address the non-determinism of peers. Third, the checkpointing with cache allows the non-determinism in peers. Finally, cache-based techniques have been extended to handle UDP protocol. It is implemented on top of JPF extension called "net-iocache".

In contrast, the concept of centralization techniques is to model check all processes within a model checker tool. The techniques can be applied at the SUT level, OS level, and model checker level. At the SUT level, the proposed works show the techniques of transforming processes into threads, how to handle class confliction, initialize and shutdown semantics, and how the local threads can communicate between each other. At the OS level, the model checker captures the entire Linux state if it detects any non-determinism within a process. Finally, at the model checker level, the work develops the multi-process JVM for JPF. The technique customizes the class loader within JPF to enable of loading local threads without any confliction of types, static functions, and fields. The internal communication between local threads has also been supported in JPF.

In summary, the cache-based approach model checks one process at a time, in which this technique can scale better than

centralization approach. However, it may miss some errors since cache-based does not cover all the network communication. In contrast, centralization approach covers all the errors since all the processes are under control of model checker. Therefore, we intend to propose a hybrid approach which combines the centralization and cache-based techniques to support verification of IoT applications.

IV. Proposed Work

Our preliminary proposal for model checking of networked applications is illustrated in Figure 7. It combines the centralization and cache-based approaches for verifying IoT applications. We are going to utilize the existing JPF extensions such as *jpf-nas* and *net-iocache*. Jpf-nas allows multi-process JVM for verifying processes within a model checker, and the net-iocache provides a special cache layer for state synchronization between the SUT and the peers. Therefore, our initial work will modify the cache layer for state synchronization between jpf-nas and the peers. It is expected that proposed work will be able to model check wider range of IoT applications.

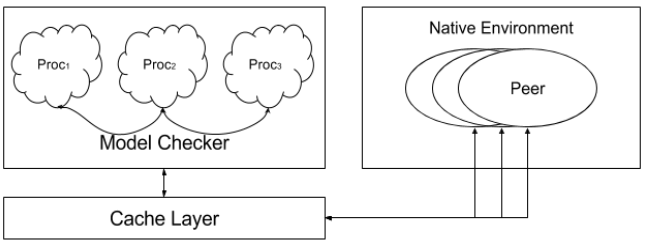

Figure 7 Architectural design of the hybrid technique by extending JPF model checker

V. Conclusion And Future Works

The reliability of IoT applications is one of the concerns for successful IoT deployment. The applications enable human-to-device and device-to-device interactions, and they need to ensure that data/messages have been received and performed properly in a timely manner. In this paper, we presented a review on the existing techniques of model checking network applications. We analyzed the architectural design of the techniques and discussed their strengths and limitations. The limitations such as the interactions between the JPF model checker with peer processes, which haven't studied before. Finally, our future work concentrates on developing JPF extension that combines centralization and cache-based for verification of IoT applications.

References

[1] T. Priestley, "The internet of things is a fragmented $19 trillion roulette gamble," Forbes, 2015. Available at https://www.forbes.com/sites/theopriestley/2015/10/05/the-internet-of-things-is-a-fragmented-19-trillion-roulette-gamble/#6f7c579d29d9. [Accessed: 28-Feb-2017].

[2] B. L. Risteska Stojkoska and K. V. Trivodaliev, "A review of Internet of Things for smart home: challenges and solutions," *J. Clean. Prod.*, vol. 140, pp. 1454–1464, Jan. 2017.

[3] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," *Bus. Horiz.*, vol. 58, no. 4, pp. 431–440, Jul. 2015.

[4] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, Cambridge, 2008.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, Cambridge, 1999.

[6] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi, "Modular software model checking for distributed systems," *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 483–501, May 2014.

[7] N. Shafiei, and P. Mehlitz, "Extending JPF to verify distributed systems," *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–5, Feb. 2014.

[8] N. Sebih, F. Weitl, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Software model checking of udp-based distributed applications," in *2014 Second International Symposium on Computing and Networking*, 2014, pp. 96–105.

[9] E. M. Clarke, "The birth of model checking," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008, vol. 5000 LNCS, pp. 1–26.

[10] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 174–186.

[11] J. C. Corbett et al., "Bandera: extracting finite-state models from java source code," in *Proceedings of the 22Nd International Conference on Software Engineering*, 2000, pp. 439–448.

[12] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and cartesian abstraction for model checking C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria, and W. Yi, Eds. Berlin, Heidelberg: Springer, 2001, pp. 268–283.

[13] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Berlin, Heidelberg: Springer, 2001, pp. 260–264.

[14] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[15] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of software components in C," in *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 385–395.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer, 2003, pp. 235–239.

[17] C. Artho, C. Artho, A. Biere, P. Eugster, M. Baur, and B. Zweimüller, "JNuke: efficient dynamic analysis for Java," in *Proc. CAV* '04, pp. 462-465, 2004.

[18] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby, "Building your own software model checker using the bogor extensible model checking framework," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer, 2005, pp. 148–152.

[19] C. Artho, B. Zweimüller, A. Biere, E. Shibayama, and S. Honiden, "Efficient model checking of applications with input/output," in *Computer Aided Systems Theory*, R. Moreno-D'iaz, F. Pichler, and A. Quesada-Arencibia, Eds. Berlin, Heidelberg: Springer, 2007, pp. 515–522.

[20] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe, "Efficient model checking of networked applications," in *46th International Conference Objects, Components, Models and Patterns Proceedings*, 2008, vol. 11, pp. 22–40.

[21] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Cache-based model checking of networked applications: from linear to branching time," in *24th {IEEE/ACM} International Conference on Automated Software Engineering*, Auckland, New Zealand, 2009, pp. 447–458.

[22] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model checking distributed systems by combining caching and process checkpointing," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 103–112.

[23] S. D. Stoller and Y. A. Liu, "Transformations for model checking distributed Java programs," in *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, 2001.

[24] C. Artho and P.-L. Garoche, "Accurate centralization for applying model checking on networked applications," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 177–188.

[25] E. Barlas and T. Bultan, "Netstub: a framework for verification of distributed Java applications," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 24–33.

[26] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato, "Model checking of multi-process applications using SBUML and GDB," in *Proc. Workshop on Dependable Software: Tools and Methods*, 2005, pp. 215–220.

[27] L. Ma, C. Artho, and H. Sato, "Analyzing distributed Java applications by automatic centralization," in *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, 2013, pp. 691–696.

[28] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–12.