Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 96 (2016) 1341 – 1350

Procedia Computer Science

20th International Conference on Knowledge Based and Intelligent Information and Engineering Systems, KES2016, 5-7 September 2016, York, United Kingdom

Testing environment for CPS by cooperating model checking with execution testing

Takeru Kuroiwa[a,b*], Yusuke Aoyama[b], Noriyuki Kushiro[b]

[a]*Mitsubishi Electric Corporation, 5-1-1, Ofuna, Kamakura, Kanagawa, 247-8501, Japan*
[b]*Kyushu Institute of Technology, 680-4, Kawazu, Iizuka, Fukuoka, 820-8502, Japan*

Abstract

In this study, we propose a testing environment for cyber-physical systems (CPS). In system testing for CPS, many tests are difficult to design or implement because of these systems' many product variations. The proposed environment executes the tests and guarantees that these systems operate reliably using two methods. The first method provides easy management of test cases by managing functions to be tested and configurations to be tested separately. The second method involves automatic testing of real devices based on model checking technologies. The authors have developed a horizontal prototype of the proposed environment and confirmed its feasibility and applicability.

Peer-review under responsibility of KES International

Keywords: Cyber-physical systems ; System testing ; Model cheking ;

1. Introduction

Recently, global, rapid product development has become a requirement for businesses to remain competitive. Developers of cyber-physical systems (CPS)[1,2] have also applied new techniques for improving operational efficiency such as OSS software and software product lines (SPLs)[3]. A mapping study[2] suggests that SPLs are used in industry to archive more efficient software development; research exists that examines the testing side of SPLs.

However, in system testing for CPS, improving the operational efficiency and ensuring the quality of the systems remain difficult for the following reasons:

- Many product variations exist because several types of devices coexist in the same system. Therefore, testers spend a great deal of time validating each variation to guarantee that all of the systems operate reliably.
- Testing conflicts among functions is extremely difficult, and thus, systems might not be adequately tested.

Peer-review under responsibility of KES International
doi:10.1016/j.procs.2016.08.179

In this study, we propose a system testing environment to solve these problems and to improve the operational efficiency and quality of testing of CPS. The proposed environment has two concepts as follows:

- Easy management of test cases. The proposed environment makes easy to manage test cases for many product variations using a repository for managing functions to be tested and configurations to be tested separately.
- Automatic testing. The proposed environment enables tests that are difficult to design or execute using model checking technologies.

The remainder of this paper is organized as follows. A brief description of the problems in the testing of CPS is presented in Section 2. Section 3 presents overviews of model checking technologies as a background. Section 4 details the architecture and implementation methods of the proposed environment; its feasibility and applicability are confirmed via an evaluation using a prototype for an air conditioning system in Section 5. Related work is discussed in Section 6. Conclusions and future studies are presented in Section 7.

2. Research problems

2.1. The problem of operational efficiency

In CPS, problems related to operational efficiency seem to result from the extensive large number of test cases involved in system testing. (For the purposes of this study, system testing is defined as the testing of a function's capability, and where the function needs more than two devices.) CPS may have many product variations so as to meet customer requirements; testers should validate all variations.

The calculation of the number of product variations is shown in the equations that follow. When the target system consists of n kinds of devices and from one 1 to as many as m of each device may exist, the number of product variations is:

$$\prod_{k=1}^{n} m_k \tag{1}$$

Thus, the number of test cases may be several times greater than the result of Eq. (1).

2.2. The problem of quality assurance

Assuring the quality of CPS is critical, because many devices in these systems are responsible for the essentials of life and must, therefore, be highly reliable.

However, in CPS, we estimate that problems remain concerning quality assurance because of the large number of functions and their asynchrony. Functions are activated or deactivated by many events that occur asynchronously such as communication, operation, and noise. These factors might cause conflicts between functions; testing these conflicts is extremely difficult. Therefore, the systems might not be adequately tested, and thus, be at risk of failure.

We assume that two functions, both of which are activated for 10 ms by external asynchronous events. The functions may conflict with each other if both events occur in a given 10 ms window, as shown in Figure 1.

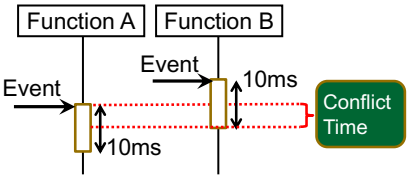

Fig. 1. Example of function conflict.

3. Background

This paper presents a system testing environment using model checking technologies. This section provides brief overviews of these technologies.

3.1. Model checking

Model checking[4] is a technique for verifying the finite state of concurrent systems. Because model checking can be performed automatically, it is preferable to deductive verification. Model checking has been a topic of extensive theoretical research for the past thirty years[5].

In model checking, the specifications of the system are expressed as temporal logic formulas. Efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds. Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula p and a model M with initial state s, decide if

$$M, s \models p \tag{2}$$

3.2. SPIN model checker

The SPIN (Simple PROMELA INterpreter) model checker[6] is an open-source software verification tool. The tool can be used for formal verification of multi-threaded software applications.

The procedure for verification using SPIN is shown in Figure 2. In SPIN, the models are written in PROMELA (PROcess MEta LAnguage) and the properties to be verified are expressed in LTL (linear temporal logic)[7]. SPIN generates a model checker code written in C from the models and the properties, which is then compiled into a model checker program. Testers obtain the results of exhaustive verification in a short time by executing the compiled program.

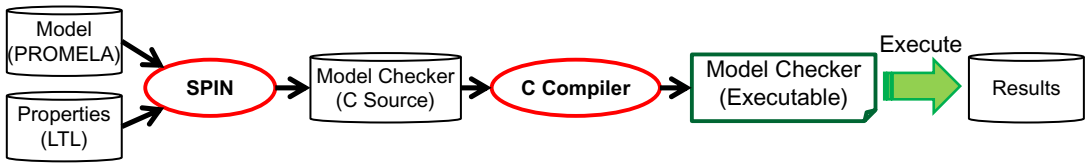

Fig. 2. Procedure for verification using SPIN.

PROMELA describes non-deterministic sequences and the model checker program performs fast (but exhaustive) verification of the model by iteratively performing all of the possible sequences. The models written in PROMELA have processes called “Proctypes” that are executed in parallel. Message channels are used to model the transfer of data from one process to another. The non-deterministic order or timing of messages is written as a PROMELA control flow.

4. Methodology

4.1. Overview

We propose two concepts to solve the previously mentioned problems of CPS, i.e., 1) easy management of test cases, and 2) automatic testing. In addition, we propose a testing environment that incorporates these concepts.

Figure 3 shows the architecture of the proposed testing environment. This architecture contains a repository to manage system functions to be tested and system configurations to be tested separately. The repository greatly declines the number of data to be managed from the product of the number of the system functions by the number of

the system configurations to the sum of them. Next, the environment automatically generates test cases from certain system functions and system configurations for the device under test, and then, carries out testing. Testers obtain the results of system testing for the device with few operations and validate product variation for the configuration. Moreover, the architecture aids in improving efficiency to build the environment because it requires only the device under test, and testers do not have to prepare as many devices as the maximum configuration of the system. Implementation methods for each concept are presented in the following sections.

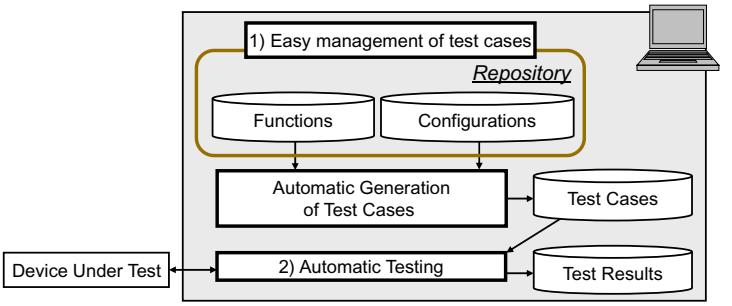

Fig. 3. Architecture of proposed testing environment.

4.2. Easy management of test cases

4.2.1. Architecture

Figure 4 (a) shows the software architecture related to easy management of test cases in the proposed environment. In this environment, test cases are generated by a template engine because each system function is written in PROMELA (mentioned in Section 3) as a source template and each system configuration is written in XML, which is the data model for the template engine.

4.2.2. Notation of system functions

In this study, the source templates of system functions contain descriptions of the communication specifications of the system. PROMELA is suitable language because order or timing for sending communication commands is non-deterministic in many CPS.

Figure 4 (b) shows the notation of the source templates. The template describes the non-deterministic order of device activation using Proctypes in PROMELA. Each Proctype corresponds to a device and contains data indicating the device type. By contrast, each instance of communication between devices is in the form of a message between Proctypes; the non-deterministic order or timing of commands is written in a PROMELA control flow.

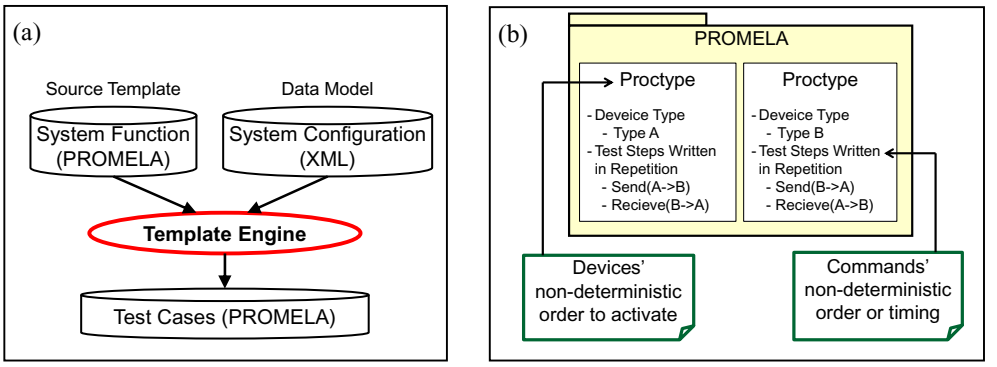

Fig. 4. (a) Software architecture related to easy management of test cases; (b) Notation of source templates to describe system functions.

4.2.3. Procedure for generating test cases

In this section, we describe the procedure of generating test cases from both the system functions and the system configurations.

The template engine shown in Figure 4 (a) instantiates the source template that corresponds to the function to be tested using one of the system configurations and generates the executable test case written in PROMELA. Specifically, the template engine duplicates each Proctype for all devices written in the system configuration. In addition, the template engine determines both the source address and the destination address of each command.

Figure 5 presents a conceptual diagram of the executable test case of one device of type A and three devices of type B generated by the proposed environment.

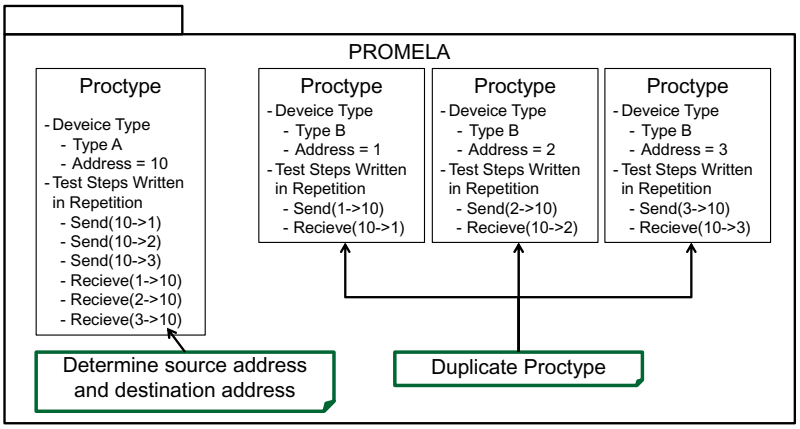

Fig. 5. Executable test case generated by the template engine of the proposed environment.

4.3. Automatic testing

4.3.1. Architecture

Figure 6 (a) shows the software architecture related to automatic testing in the proposed environment. This architecture includes a modified version of SPIN, a GUI to allow testers to easily analyze test results, and a driver of network interface devices used to communicate with the real devices under test.

A modified version of SPIN generates a model checker program to iteratively perform all possible communication sequences described in the system testing cases, some of which are difficult for testers to design or execute. The model checker program automatically runs and generates the test results.

4.3.2. Procedure for automatic testing

The procedure for automatic testing within the proposed testing environment (Figure 6 (b)) is described as follows:

- Testers select a test case and a device to be tested from the devices written as Proctypes in the test case.
- The model checker program executes the test case and sends the communication commands, written as Proctypes (except that corresponding to the device under test), via the network interface device.
- The model checker program receives the communication commands from the device under test. The program then determines the results of the test case by comparing the communication commands received from the device under test to those written in Proctype which correspond to the same device.
- Testers analyze the test results and communication log using the analysis support GUI if the test results are considered unacceptable.

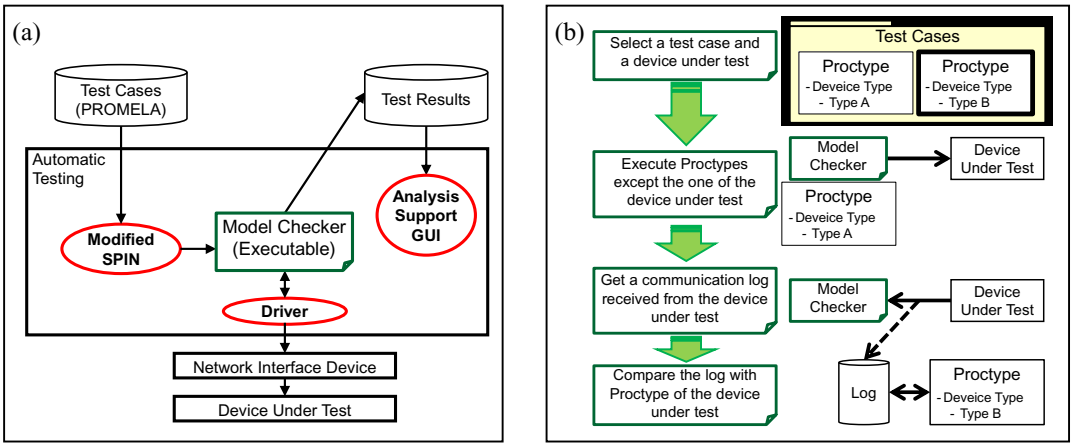

Fig. 6. (a) Software architecture for automatic testing; (b) Procedure for automatic testing.

4.3.3. Cooperating model checking with execution testing

We determined that SPIN should be modified to perform communication with a real device in the proposed environment, because the model checker program generated by the original (unmodified) SPIN performs communication between only those processes running in the program and iteration of only non-deterministic sequences. In system testing, each time, the initial settings for the device under test are required. However, these initial settings are deterministic and the original SPIN does not iteratively change the initial settings. Therefore, the following modifications have been made:

- Communicate with real devices using a dedicated message channel
- Iteratively evaluate not only non-deterministic sequences but deterministic sequences (see Figure 7)

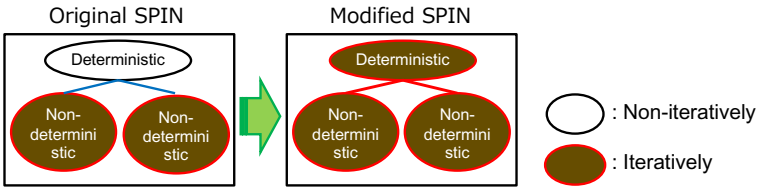

Fig. 7. Modification of SPIN.

5. Empirical Study

We have confirmed the proposed environment's feasibility and applicability in an evaluation using a prototype for an air conditioning system. This section presents overviews of both the target system and the prototype, and shows the results of the evaluation.

5.1. Target system

An air conditioning system has some part of control systems' features of CPS shown in a survey[8] as follows, and thus, the system is adequate for the evaluation of the proposed environment.

- The system consists of several kinds of physical devices with limited resources connected to each other via a dedicated network for the system. (see Figure 8 (a))
- The new functions of the system are implemented during the software upgrade of each device. (see Figure 8 (b))

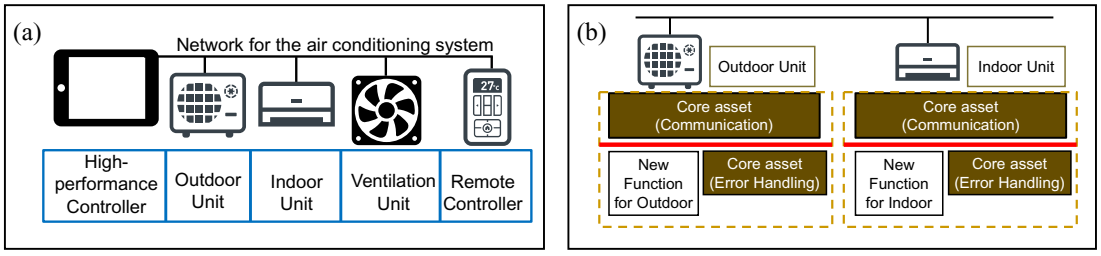

Fig. 8. (a) Configuration of the target air conditioning system; (b) Software architecture of the air conditioning devices.

5.2. Prototyping of proposed environment

Figure 9 (a) shows the hardware architecture of the prototype. The air conditioning device under test is connected to the prototype on a PC via a network interface device for interconversion between the communication protocol of the air conditioning system and the USB protocol.

The software architecture of the prototype is shown in Figure 9 (b). The prototype includes the template engine, the modified SPIN, the model checker program, and the driver mentioned in Section 4. The architecture also includes a graphical user interface (GUI) to accept inputs for the template engine and to analyze the results which the model checker program outputs.

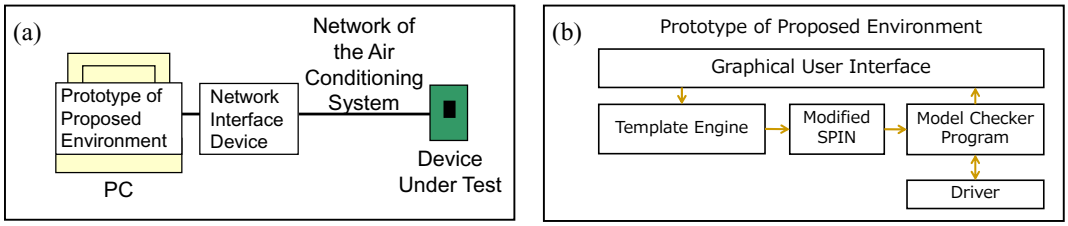

Fig. 9. (a) Hardware architecture of the prototype; (b) Software architecture of the prototype.

Figure 10 (a) shows the prototype's GUI used to make system configurations. Testers can configure the system by means of mouse operations to place blocks which represent each device. Testers can also use the keyboard to input the parameters of each unit, such as device type and address. Figure 10 (b) shows the GUI used in executing the tests. Testers easily select the device under test by checking boxes in the "isVirtual" column. Testers can also obtain test results easily by simply checking the field showing "Success" or "Failure".

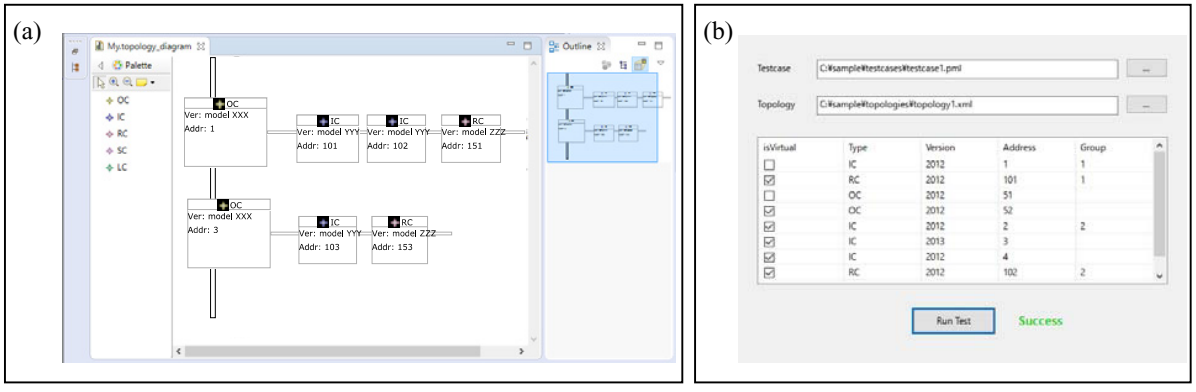

Fig. 10. (a) GUI used to make system configurations; (b) GUI used to execute the test.

5.3. Evaluation

We evaluated the feasibility and applicability of two concepts of the proposed environment (mentioned in Section 1) through some test cases of an air conditioning system.

5.3.1. Test cases for evaluation

We attempted to test two cases shown in Table 1 using the prototype. The test cases took much time to design and implement in developing the air conditioning system.

Table 1. Test cases for the evaluation of the proposed environment.

No.	Test item name	Point of test	Difficulty to test
(i)	Change Mode of Ventilation Unit	The mode of the ventilation unit is changed regardless of the order of the communication	To test various orders of the communication and various types of the ventilation unit
(ii)	Conflict Scheduled Operations	The one scheduled operation is NOT activated if the other schedule is activated	To induce conflict deliberately

5.3.2. Results and discussion

Table 2 shows the results of the evaluation about easy management of test cases and automatic testing.

Table 2. Results of the evaluation.

No.	Easy management of test cases	Automatic testing
(i)	Good	Good
	We could describe the communication for changing the mode in a source template and the various configurations in XML. Additionally, the template engine output test cases of every order of the communication.	The model checker program generated by the modified SPIN automatically executed every order of the communication to the ventilation unit.
(ii)	Improvement is required	Good
	It remained difficult to make a source template corresponding to each scheduled operation. We had to describe the communication of two scheduled operations in one source template in order to induce conflict.	Same as above.

Figure 11 (a) and Figure 11 (b) are the sequence diagrams which display the communication log during the test (i) generated by the prototype. These diagrams show that the communication commands that send to or receive from the real ventilation unit under test in different order controlled by the model checker program. Moreover, via the GUI, testers could analyze a test failure in a short time, and thereby, improve operational efficiency.

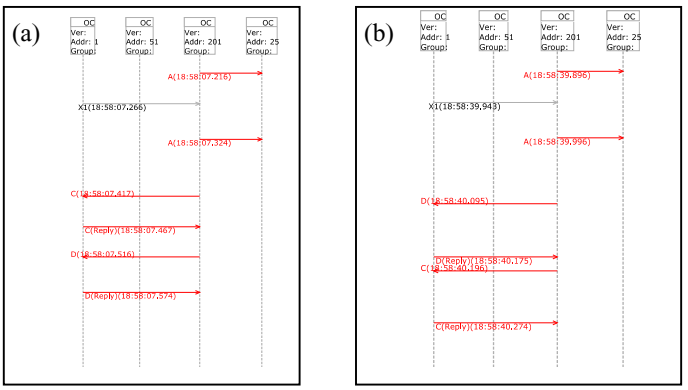

Fig. 11. (a) One order of the communication during the test (i); (b) Another order of the communication during the test (i).

Therefore, we determined that the concept of the proposed testing environment is feasible and beneficial.

However, tests of function conflicts remain difficult in consideration of the result of test (ii). We would improve the template engine to combine two or more source templates which describe each function, and to alternate between deterministic and non-deterministic sequences for each communication sequence for testers to set sequences which may be a factor in these conflicts to non-deterministic, while setting other sequences to deterministic. Figure 12 shows the conceptual diagram of GUI used to alternate between deterministic and non-deterministic sequences.

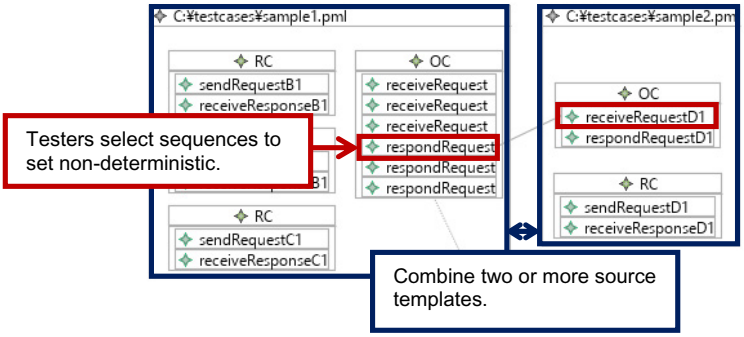

Fig. 12. GUI used to alternate between deterministic and non-deterministic sequences.

6. Related works

Algorithms of conformance testing between models and their implementations of CPS[9,10] were suggested. For formal modeling and verification of CPS, a framework to support testing and analysis[11] was developed.

On the other hand, model checking methodologies for web applications[12] and for embedded systems[13] have been proposed. An automation method of generating models for model checking[14] was also suggested. A case study[15] shows that an experimental fault analysis process using model extraction and model checking can detect the cause of failures that are hard to reproduce. Our proposed testing environment leads to enable testers to detect such failures on real devices.

Moreover, a framework combining model checking and conformance tests[16] enabled fully-automatic verification of embedded real-time systems and improved operational efficiency. Our study aims to achieve this goal in system testing for real embedded devices in CPS.

7. Conclusion

In this study, we described the concepts and problems related to system testing of CPS. One major problem concerns the operational efficiency of system testing; a second problem concerns methods of ensuring the quality of the systems being tested.

We proposed a system testing environment to solve these problems. We developed and evaluated prototypes that utilize the following methods: easy management of test cases and automatic testing. These prototypes were shown to improve the operational efficiency. In addition, we confirmed through testing and evaluation that our prototype using modified SPIN would enable the tests for the real devices in various orders of the communication that was difficult to execute.

The following tasks remain to be addressed in future studies:

- Architecture and management method of the repository. We proposed a repository to manage system functions and system configurations for testing. We will examine the architecture and management methods used to further improve operational efficiency.

- Application to product development. Our objective is to apply the proposed testing environment to product development. We hope to evaluate the testing environment when implemented in development sites and then refine its scope and performance.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Numbers JP16721570, and CREST, JST. The support and comments of S. Suzuki were of considerable help in developing the prototype of our proposed testing environment.

References

1. Lee, Edward A. Cyber physical systems: Design challenges. *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on.* IEEE, 2008. p. 363-369.
2. Rajkumar, Ragunathan Raj, et al. Cyber-physical systems: the next computing revolution. *Proceedings of the 47th Design Automation Conference.* ACM, 2010. p. 731-736.
3. Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns. *Addison-Wesley,* Boston, MA. 2001.
4. Clarke, E. M., Grumberg, O., and Peled, D. Model Checking. *MIT Press,* Cambridge, MA. 1999.
5. Alur, R., Henzinger, T. A., and Vardi, M. Y. Theory in practice for system design and verification. *ACM Siglog News*. 2015. 2.1, p. 46-51.
6. Holzmann, G. J. The model checker SPIN. *IEEE Transactions on software engineering*. 1997. 5, p. 279-295.
7. Pnueli, A. The temporal logic of programs. *Foundations of Computer Science.* 1977. p. 46-77.
8. Shi, Jianhua, et al. A survey of cyber-physical systems. *Wireless Communications and Signal Processing (WCSP), 2011 International Conference on.* IEEE, 2011. p. 1-6.
9. Woehrle, M. et al. Conformance testing for cyber-physical systems. *ACM Transactions on Embedded Computing Systems.* 2012. 11, p. 1–23.
10. Abbas, H. et al. Conformance Testing as Falsification for Cyber-Physical Systems. *arXiv.* 2014.
11. Buzhinsky, I., Pang, C., & Vyatkin, V. Formal Modeling of Testing Software for Cyber-Physical Automation Systems. *Trustcom/BigDataSE/ISPA, 2015 IEEE.* IEEE, 2015. 3, p. 301-306.
12. Homma, Kei, et al. Modeling, verification and testing of web applications using model checker. *IEICE transactions on information and systems.* 2011. 94.5, p. 989-999.
13. Kim, Y and Kim, M. SAT-based Bounded Software Model Checking for Embedded Software: A Case Study. *Software Engineering Conference (APSEC).* 2014. p. 55-62.
14. Konoshita, R and Sakurai, K. Model generation by the exhaustive search for embedded assembly programs and application to model checking. *Consumer Electronics (GCCE).* 2014. p. 699-702.
15. Ogawa, H et al. Experimental Fault Analysis Process Implemented Using Model Extraction and Model Checking. *Computer Software and Applications Conference (COMPSAC).* 2015. p. 95-104.
16. Herber, P and Glesner, S. Verification of Embedded Real-time Systems. *Formal Modeling and Verification of Cyber-Physical Systems.* Springer Fachmedien Wiesbaden; 2015. p. 1-25.