Approach to Testing Many State Machine Models in Education

Shinpei Ogata, Mizue Kayama and Kozo Okano
Faculty of Engineering, Shinshu University, Wakasato 4-17-1, Nagano-shi, Nagano, 380-8553, Japan

Keywords: Domain-Specific Language, Modeling Education, Simulation, State Machine Diagram, Testing, Unified Modeling Language.

Abstract: In state machine modeling education, the effort required by instructors to test a large number of learner-created models should be reduced to concentrate on the following feedback activity. Although there are several methods for validating and verifying a state machine model, a considerable problem for instructors is the lack of tools to test multiple models at once. This study proposes a preliminary approach and a support tool to efficiently and promptly test multiple state machine models. A basic approach to solving this problem is creating test cases and then testing multiple state machine models simultaneously using these test cases. To reduce the instructors' testing effort, the proposed approach includes three new concepts: (1) a logger extension to capture simulated data generated by an existing state machine simulation tool called SMart-Learning; (2) a method for creating test cases based on these logs; and (3) a feature to test many models using these test cases. As a result of a preliminary evaluation, it was confirmed that the proposed approach could be useful to test many answer models efficiently.

1 INTRODUCTION

Modeling education has become more important by entering the Cyber-Physical Systems (CPSs) era. Large scale CPSs often include geo-distributed physical components (Hu et al., 2019; Peng and Ho, 2018; Tranoris et al., 2018; Gupta and Ramachandran, 2018; D'souza and Rajkumar, 2017; Saurez et al., 2016; Khalaf and Abdalla, 2016). Developers find hard to test these physical components due to the cost and limitation of their deployment. Thus, the logical validation and verification of such CPSs have become increasingly important.

The state machine diagram notation provided by Unified Modeling Language (UML) (Object Management Group, 2017b) and OMG System Modeling Language (SysML) (Object Management Group, 2017a) is often used to define the life-cycle of CPS components (Graja et al., 2018; Vidal and Villota, 2018; Pencheva and Atanasov, 2016). Furthermore, methods to validate and verify state machine diagrams (itemis AG, 2019; Bagherzadeh et al., 2017; Das et al., 2016) and convert state machine diagrams into models used by model checking tools (Damjan and Vatanawood, 2017; Nobakht and Truscan, 2013) have been proposed in the past. Model checking tools such as SPIN (Holzmann, 1997) and UPPAAL (David et al., 2015) can exhaustively check whether a system model meets its specifications. Therefore, it is important for developers to learn state machine modeling.

It is difficult for novices to correctly understand and perform state machine modeling, even if there is a checking environment, owing to the lack of their knowledge regarding the model or the checking environment (Ogata et al., 2017). Therefore, the instructors must check learner-created models to give feedback to the learners. Although there are methods to make the efficient validation or verification of state machine diagrams (itemis AG, 2019; Bagherzadeh et al., 2017; Das et al., 2016; Ogata et al., 2017), these methods cannot be applied to multiple models simultaneously. Thus, the instructors need to perform check operations repeatedly for the number of learners.

To improve this problem, this study proposes a preliminary approach to test a large number of state machine models created by learners, and also a support tool extending the educational model simulation tool called SMart-Learning (Ogata et al., 2017). A basic idea to solve the problem is to create test cases with which multiple state machine models are tested simultaneously. To reduce the instructors' testing effort, the proposed approach includes three new concepts: (1) a logger extension to capture simulation data generated by the SMart-Learning tool; (2) a method to create test cases based on these logs; and

Ogata, S., Kayama, M. and Okano, K.
Approach to Testing Many State Machine Models in Education.
DOI: 10.5220/0007768004810488
In *Proceedings of the 11th International Conference on Computer Supported Education (CSEDU 2019)*, pages 481-488
ISBN: 978-989-758-367-4
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

(3) a feature for testing many models using the test cases. As a result of a preliminary evaluation, the possibility that the proposed approach is useful to test many answer models efficiently was confirmed.

The rest of the paper is organized as follows. Section 2 discusses the problems of state machine modeling in learning. Section 3 presents the characteristics of state machine diagrams used in this study. Section 4 proposes a preliminary approach to test a large number of state machine models efficiently, and introduces a tool extending the SMart-Learning tool. Section 5 reports the result of a preliminary evaluation regarding the degree that the proposed approach can predict each correctness of many state machine models answered by learners. Section 6 discusses the effectiveness of the proposed method and future work. Section 7 compares the proposed approach with related work and, finally, we conclude the paper in Section 8.

2 PROBLEMS OF STATE MACHINE MODELING IN LEARNING

Novice modelers usually do not precisely understand the notation of state machine models when they start to learn. The novices start to understand the notation by reading textbooks or by listening to lectures regarding the subject. However, when they want to know whether their own model is problematic, in most case, the only way they have is to hear the specialists such as the instructors regarding the model. In (Ogata et al., 2017), the educational state machine simulation tool called SMart-Learning is proposed, to assist such learners to self-check the models. However, when the learners cannot understand precisely the usage or output of the visual simulation, the tool is difficult to assist them to perform self-checking. Therefore, it is still important for the learners to receive feedback on their model from the instructors.

However, the instructors' effort of checking a large number of learner-created models is considerable and very time-consuming. It is very hard for the instructor to concentrate on the following feedback activity. Although there are some methods for validating and verifying a state machine model(itemis AG, 2019; Bagherzadeh et al., 2017; Das et al., 2016; Ogata et al., 2017), these methods do not provide a feature to test multiple models at once, and the instructors have to repeat these operations for the number of learners. A basic idea to improve this problem is to create test cases and test multiple state machine models at once using these test cases. Therefore, we aim to establish a method of testing taking the ease of creation and test into account. A key new idea is to reuse simulation logs outputted by the SMart-Learning tool in order to create executable test cases. A merit of this idea is that not only instructors but also learners may easily create executable test cases.

3 STATE MACHINE DIAGRAMS IN THE LECTURE

The state machine diagram is used for modeling discrete event-driven behaviors (Object Management Group, 2017b). In this lecture, the learners create extended state machine diagrams for line trace robots and then test their diagrams by running the robots with C++ programs generated from the diagrams. The extension implies introducing DSL, communication, and scripts into state machine diagrams.

Regarding the DSL, the values of *do activity*, e.g. `turnRight`, and *trigger*, e.g. `detectBlackColor`, are selected from the respective terminologies, which can be translated into code fragments, and a DSL model can be transformed into a complete program.

Regarding the communication, it synchronizes the timing of state transitions between different transitions, and is represented using messages between the senders and the receivers. A transmitted message, e.g. `in!` is written in a *effect* with an exclamation mark. A received message, e.g. `in?` is written in a *trigger* with a question mark. The alphabets must be the same between these corresponding messages.

Regarding the scripts, the value of *guard*, e.g. `count==1`, and *action*, e.g. `count++;` is written using JavaScript syntax. When such scripts are simple, we consider that the differences between languages such as JavaScript and C++ are negligible. The SMart-Learning tool automatically extracts the variables from *guard* and *effect*. These scripts, including the variables, will be embedded in the program generated directly. Hereafter, the state machine diagram implies the extended state machine diagram.

4 PROPOSED APPROACH

This section describes our proposed approach and presents an extension of the SMart-Learning tool, called SMart-Learning for Instructor (SML4I) tool.

4.1 Overview

Figure 1 shows an overview of the proposed approach. All the steps are listed as follows:

1. The instructors create the reference model of modeling tasks and then assign the tasks to learners.
2. The instructors simulate the reference model using the SML4I tool and then obtain the resulting simulation logs for creating the test case. The simulation using the SML4I tool is detailed in Section 4.2. The logger extension as a new part of our proposed approach is detailed in Section 4.3.
3. The instructors abstract the simulation logs to create the test case. The test case creation is detailed in Section 4.4.
4. The learners create their answer model of the same task and then check the model using the SML4I tool.
5. The learners submit the model to the instructors after refining the model appropriately.
6. The instructors test all the submitted models using the SML4I tool. The testing is detailed in Section 4.5.
7. The instructors analyze each model after testing them and then provide feedback to the learners.

4.2 Model Simulation

The model simulation features of the SML4I tool have already been concluded in (Ogata et al., 2017), and are explained briefly in this paper. The guideline of the SML4I tool is that learners can concentrate on modeling state machines and make it easy to use in education. The SML4I tool takes a few easy steps to start to simulate a state machine model immediately after creating only state machine diagrams.

Figure 2 shows the feature to prepare the model simulation, called State Machine Instance Configurator (SMIC). The "QUICK SETTING" function automatically generates all classes as the context of state machine diagrams and attributes as variables. Furthermore, the SMIC feature can create multiple instances of a state machine diagram. In Fig. 2, `PriorityCar.state1:PriorityCar.state` shows one instance of the state machine diagram `PriorityCar.state`.

The SML4I tool provides three types of model simulators. However, only the most basic simulator will be described because it is sufficient to explain the new part of the tool. Figure 3 shows the State Machine Manual Simulator (SMMS). By using the SMMS function, the users can input a sequence of events to make the state machine instances behave. The users can select an event from the buttons at the bottom left panel in Figure 3 and input it to state machine instances.

The state machine instance under simulation is shown in the top view of Figure 3. The states and transitions filled with red indicate the current state and the latest transition, respectively. The bottom center panel displays the values referred by the variables used in the scripts. The values are automatically updated by accessing the Java ScriptEngine instance whenever inputting an event to the state machine instances. The syntax of an event is <event name><instance scope> The instance scope determines state machine instances to provide to the event. The <instance scope> expression is further categorized into three types: <all instances>, i.e. "", <a type of instances>, e.g. "::NormalCar.state", or <an instance>, e.g. "::NormalCar.state1".

4.3 Logger Extension

This logger extension captures three types of logs: instance configuration, event logs, and transition logs. The left side of Figure 4 shows an example of simulation logs. The list of `[configuration]` specifies what instances are needed in the test. The instance configuration corresponding to the list of `[configuration]` is captured when the state machine instances are created using the SMIC function.

The list of `[events]` specifies a sequence of events to input. The list of `[expected]` specifies the expected result consisting of one or more steps. A step consists of 6 items: <state machine instance>, <transition source>, <event>, <guard>, <effect>, <transition target>. An example is `PriorityCar.state1, rightTurn, detectBlackColor, , !, goForward`. The event and transition logs corresponding to the list of `[events]` and `[expected]`, respectively, are captured whenever inputting an event to the state machine instances during simulation.

4.4 Test Case Creation

Instructors abstract the simulation logs to create test cases because the logs are likely to be specialized to the reference model. A basic abstraction strategy is to remove the elements which may be freely determined by learners. For instance, the message names of communication, scripts, and pseudo-states, such as initial pseudo-states, are candidates to be removed. The right side of Figure 4 shows the test case extracted

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

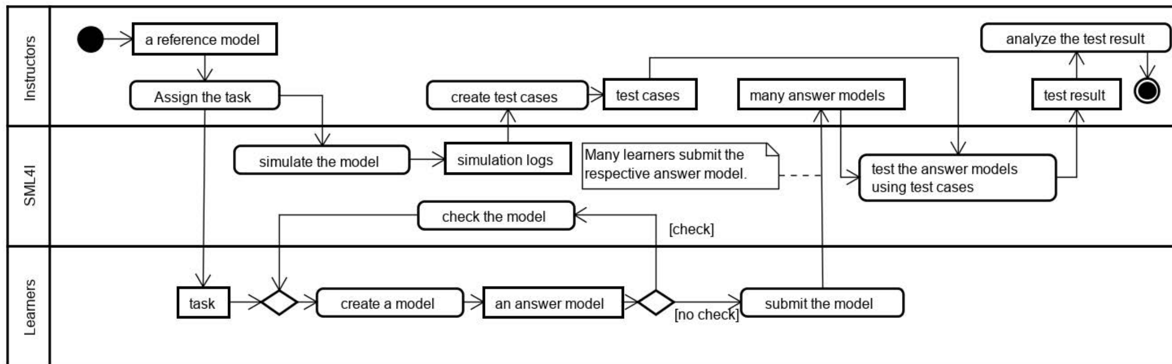

Figure 1: Overview of the proposed approach.

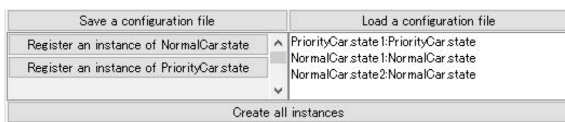

Figure 2: State machine instance configurator.

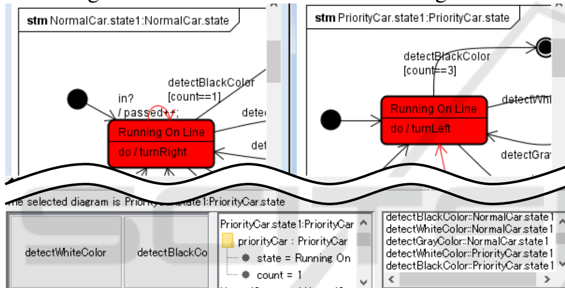

Figure 3: State machine manual simulator.

from the logs on the left side. Although there is the possibility that the SML4I tool can assist instructors to extract the simulation logs based on the strategy mentioned above, this feature will be established in a future work.

4.5 Testing

The SML4I tool tests each answer model using given test cases. The procedure to test the models is as follows.

1. The tool instantiates state machines based on the instance configuration of the test case.
2. The tool extracts the sequence of the events from the test case and inputs those events to the instances.
3. The tool checks to what extent the execution result fulfills the expected result. When the expected result is completely fulfilled, the tool assumes that the test case passed. Otherwise, the tool assumes that the test case failed.

The steps to check whether the expected result was fulfilled or not is explained in detail below. Figure 5 shows the overview of these steps.

1. Each step of an expected result is apportioned into the step group corresponding a state machine instance in order. The state machine instance can be identified by the <state machine instance> value at each step.
2. The step extracted from the latest transition of the simulator (hereafter, called a simulator step) is compared with the next step of the expected result (hereafter, called test case step) whenever inputting an event to the simulator. The step group to compare with can be identified by the <state machine instance> value. The next step can be identified by getting the next step of the last fulfilled step. The steps in each step group have to be fulfilled in order.
3. When the simulator step matches with the test case step, the test case step is assumed to be fulfilled. To check the fulfillment, the SML4I tool checks whether every item of the simulator step contain the corresponding item of the test case step. For instance, when the <event> value of a simulator step is `message?` and the <event> value of a test case step is `?`, the item of the test case step is assumed to be fulfilled, since the former contained the latter. If the simulator step does not match with the test case step, the simulator step is ignored and then Step 2 is performed for the next simulator step. Step 2-3 are performed iteratively until all simulator steps are processed.
4. When all steps in the expected result are fulfilled, the test case is assumed to pass. Otherwise, the test case is assumed to fail.

SCIENCE AND TECHNOLOGY PUBLICATIONS

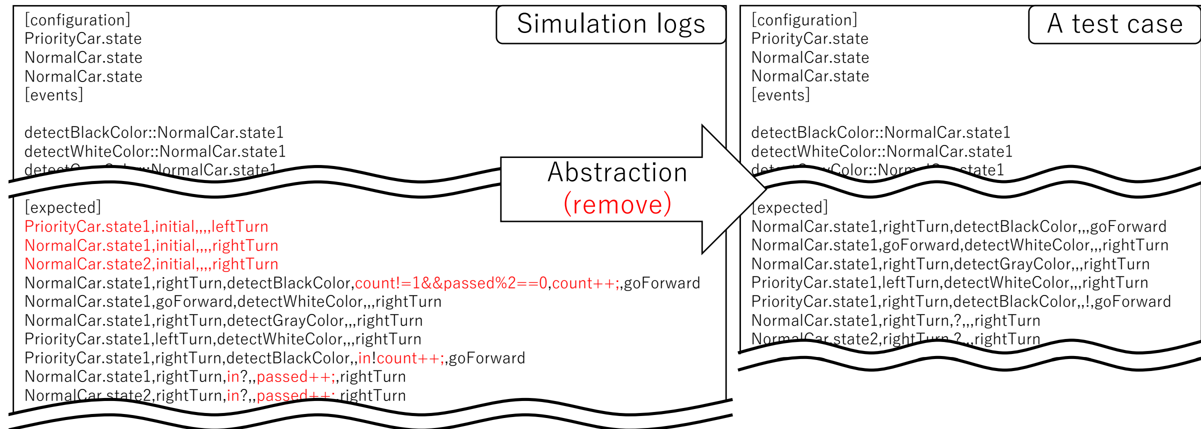

Figure 4: Abstraction of simulation logs to create a test case.

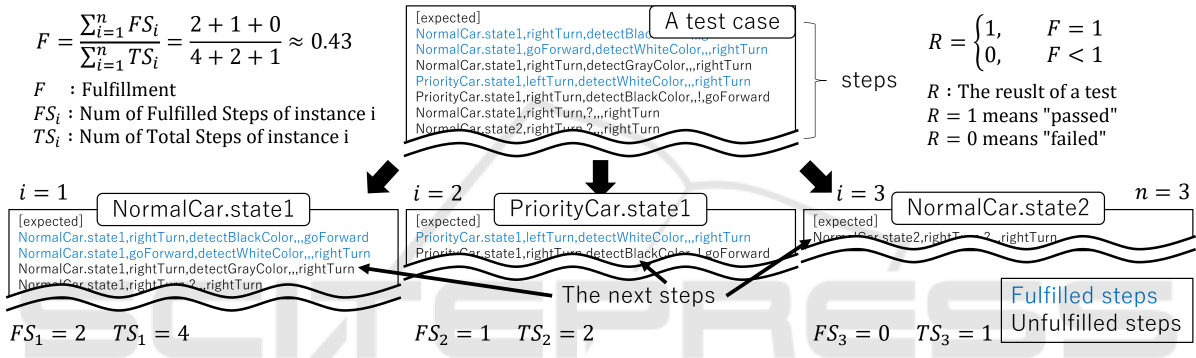

Figure 5: Overview of checking the fulfillment.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

5 PRELIMINARY EVALUATION

The proposed approach was preliminarily evaluated to determine its potential effectiveness. In this evaluation, we checked how the SML4I tool correctly predicted each of the 73 answer models.

5.1 Overview

The SML4I tool was prototyped using the model editor astah (Change Vision, 2019), and it was used to check 73 answer models created by the learners. Each of these 73 models was created to achieve the task shown in Figure 6. These robots run alternately. One robot (R1) first starts to run on a white line and then stops when it detects a gray line. R1 sends a signal to the other robot (R2) immediately after stopping. When R2 receives the signal, it starts to run on a gray line and then stops when it detects a white line. In Similarly, R1 starts to run immediately after R2 stops. After R1 detects a gray line three times, R2 completely stops and R1 starts to run backward. Then, when R1 detects the change of color, it stops

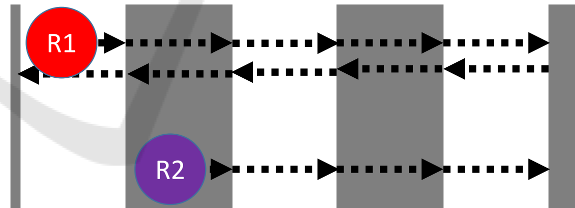

Figure 6: Image of the task of modeling two line trace robots.

for 2 seconds. After R1 detects a gray line three times, R1 completely stops.

We used one of the correct answer models as the reference model in this evaluation because we avoided decreasing the objectiveness from this evaluation and showed the ease of use of the proposed approach. We defined one test case by obtaining and abstracting the simulation logs of one simulation of the reference model. The test case was abstracted by the experimenter using the basic strategy mentioned above, and it represented the basic flow of the processes and interactions of the two robots to achieve the task.

Table 1: Confusion matrix based on the evaluation result.

		Prediction Positive	Prediction Negative	Total
Actual	Positive	TP(10)	FN(14)	24
	Negative	FP(2)	TN(47)	49
	Total	12	61	73

TP : True Positive, FP : False Positive, FN : False Negative, TN : True Negative.

The result of the evaluation is shown as a confusion matrix (Table 1). When a state machine model conforms to the notation and passes the test case, it is assumed to be positive, i.e., correct. Otherwise, a state machine model is assumed to be negative, i.e., incorrect.

5.2 Result

Table 1 shows the confusion matrix as the result of the evaluation. The recall, precision and accuracy values are calculated from the number of TP, FP, TN, and FN in the matrix and are as follows.

$$Recall = \frac{TP}{TP+FN} \approx 0.42$$

$$Precision = \frac{TP}{TP+FP} \approx 0.83$$

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN} \approx 0.78$$

On one hand, 13 out of the 73 models passed the test case, but 1 out of the 13 passed models had notation errors. As a result, the number of predicted positive models was 12. On the other hand, 24 out of the 73 models were manually evaluated as positive. Consequently, the recall value became low in comparison with the other values.

A cause of the low recall value was that the modeling strategy was varied among the correct answer models. The modeling strategy included the divisions of states and the selection of events to transit states. Consequently, the correct answer models being similar to the reference model were correctly predicted; otherwise, they were incorrectly predicted.

6 DISCUSSION

Our final objective is to pertinently identify the causes of the errors contained in all answer models so that the instructors can concentrate on the following feedback activity. As one of the fundamentals toward achieving this objective, the correctness of answer models must be correctly predicted.

The proposed approach showed a good result in terms of the precision and accuracy values because they reached approximately 80% in the evaluation. More than half of the correct models could not be predicted by one test case. Therefore, this problem was planned using multiple test cases and the threshold of the test result, because the modeling strategy was varied among the correct answer models. We plan to exhaustively analyze the answer models and consider how to create appropriate test cases to more precisely predict the correctness of the models.

As future work, to achieve the final objective, techniques to exhaustive analyze state machine models must be established. For instance, some of the major challenges are the generation and abstraction of sufficient number of adequate test cases. Regarding the generation of the test cases, MBT (Model-Based Testing) techniques are one of the effective methods to obtain test cases taking the coverage into account. Toward automating test case creation, we plan to use such techniques to efficiently create valid test cases.

Regarding the abstraction of test cases, there exists the possibility of utilizing the notation. This is a natural idea because the notation, such as DSL, is usually customized depending on the tasks. If the common definition among answer models can be identified from the notation, the simulation logs can be automatically abstracted based on the strategy mentioned above.

Additionally, to detect the causes of the errors in the incorrect answer models, we plan to analyze the checking process of the fulfillment and then propose an analysis method based on the result. One possible method to detect the causes is to define patterns for how to fulfill or not to fulfill expected results and then predict the causes based on the pattern. In addition, metrics to measure the process in which the expected results are fulfilled will also be considered.

7 RELATED WORK

Educational approaches adopting Model-Driven Development (MDD) or Model-Based Development (MBD) process including state machine modeling have been practiced in Project-Based Learning (PBL) projects (Akayama et al., 2013; Arya et al., 2017), with excellent guidelines. However, they have not provided tools to evaluate a large number of state machine models efficiently. Our study proposed an approach and the SML4I tool for improving this problem.

(Arya et al., 2017) presented test cases to test state machine diagrams. For instance, the input of such test cases comprised only events, the combination of events and states, and so on (Arya et al., 2017; Hamon et al., 2005). In addition, such test cases were generated from state machine diagrams or spreadsheets (Hamon et al., 2005; Mujjiga and Sukumaran, 2007) using a tool, e.g. SAL(Symbolic Analysis Laboratory)(Moura et al., 2004). Our study aims to propose a useful method of testing various answer models by using test cases. As far as we know, there is no study aiming the automation of testing multiple state machine models at once.

An environment using a multi-touch interface has been proposed for aiding collaborative learning of UML modeling including state modeling (Basheri et al., 2012; Basheri et al., 2013). A method for creating state machine diagrams based on an initial class diagram and texts describing class behavior have been proposed (Choppy and Reggio, 2009). A method for assessing a solution activity diagram based on a reference according to trace information has been proposed (Striewe and Goedicke, 2014). Our study aims to evaluate many state machine diagrams to efficiently improve the cost-performance of feedback creation. Thus, the objective of our approach differs from those of existing works.

8 CONCLUSION

In this paper, we proposed a preliminary approach to efficiently test a large number of answer models created by learners. In the evaluation, the result showed that the correctness of the answer models can be predicted with a high percentage of accuracy, i.e., approximately 78%. This fact suggests the possibility that problem detection for a large number of models can be efficiently performed through sufficient tests in the future.

Meanwhile, there are several major challenges remaining. As future work, we will compare the answer models with the reference model in detail, to clarify the cause of differences between the manual evaluation results and testing results. We will also analyze the process of the fulfillment of the expected results for each answer model in detail, to explore a method to predict the errors and these causes. Subsequently, we will improve the proposed approach based on the analysis results. Furthermore, we plan to use the existing MBT methods to create test cases more efficiently and utilize the state machine to abstract simulation logs more efficiently.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Numbers JP16H03074, and JP16K00094.

REFERENCES

Akayama, S., Kuboaki, S., Hisazumi, K., Futagami, T., and Kitasuka, T. (2013). Development of a modeling education program for novices using model-driven development. In *WESE'12, Workshop on Embedded and Cyber-Physical Systems Education*, pages 4:1–4:8. ACM.

Arya, K., Coelho, B., and Pandya, S. (2017). A model based design approach to system building using the e-yantra educational robot. *SIGBED Rev.*, 14(1):37–43.

Bagherzadeh, M., Hili, N., and Dingel, J. (2017). Model-level, platform-independent debugging in the context of the model-driven development of real-time systems. In *ESEC/FSE'17, 11th Joint Meeting on Foundations of Software Engineering*, pages 419–430. ACM.

Basheri, M., Burd, L., and Baghaei, N. (2012). In *OzCHI'12, 24th Australian Computer-Human Interaction Conference*, pages 30–33. ACM.

Basheri, M., Munro, M., Burd, L., and Baghaei, N. (2013). Collaborative learning skills in multi-touch tables for uml software design. *International Journal of Advanced Computer Science and Applications*, 4(3):60–66.

Change Vision (2019). Astah. http://astah.net/.

Choppy, C. and Reggio, G. (2009). A method for developing uml state machines. In *SAC'09, ACM Symposium on Applied Computing*, pages 382–388. ACM.

Damjan, P. and Vatanawood, W. (2017). Translating uml state machine diagram into promela. In *IMECS'17, International MultiConference of Engineers and Computer Scientists*, volume I. IAENG.

Das, N., Ganesan, S., Jweda, L., Bagherzadeh, M., Hili, N., and Dingel, J. (2016). Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In *MODELS'16, 19th International Conference on Model Driven Engineering Languages and Systems*, pages 36–43. ACM/IEEE.

David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Poulsen, D. B. (2015). Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415.

D'souza, S. and Rajkumar, R. R. (2017). Time-based coordination in geo-distributed cyber-physical systems. In *HotCloud'17, 9th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX Association.

Graja, I., Kallel, S., Guermouche, N., Cheikhrouhou, S., and Hadj Kacem, A. (2018). A comprehensive survey on modeling of cyber-physical systems. *Concurrency and Computation: Practice and Experience*, page e4850.

Gupta, H. and Ramachandran, U. (2018). Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *DEBS'18, 12th ACM International Conference on Distributed and Event-based Systems*, pages 148–159. ACM.

Hamon, G., De Moura, L., and Rushby, J. (2005). Automated test generation with sal. CSL Technical Note, SRI International.

Holzmann, G. J. (1997). The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295.

Hu, F., Liu, Q., Wu, J., and Yao, J. (2019). Minimizing geo-distributed interactive service cost with multiple cloud service providers. *IEEE Access*, 7:3320–3335.

itemis AG (2019). Yakindu statechart tools. Retrieved from https://www.itemis.com/en/yakindu/statechart-tools.

Khalaf, A. A. and Abdalla, A. H. (2016). Analysis of cost minimization methods in geo-distributed data centers. In *ICCCE'16, International Conference on Computer and Communication Engineering*, pages 241–245. IEEE.

Moura, L. d., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., and Tiwari, A. (2004). Sal 2. In Alur, R. and Peled, D. A., editors, *Computer Aided Verification*, pages 496–500. Springer Berlin Heidelberg.

Mujjiga, S. and Sukumaran, S. (2007). Modelling and test generation using sal for interoperability testing in consumer electronics. In *AFM'07, Second Workshop on Automated Formal Methods*, pages 32–40.

Nobakht, M. and Truscan, D. (2013). Tool support for transforming uml-based specifications to uppaal timed automata. Technical Report 1087, TUCS.

Object Management Group (2017a). OMG system modeling language 1.5. Retrieved from https://www.omg.org/spec/SysML/1.5/PDF.

Object Management Group (2017b). Unified modeling language 2.5.1. Retrieved from https://www.omg.org/spec/UML/2.5.1/PDF.

Ogata, S., Kayama, M., and Okano, K. (2017). SMart-Learning: State machine simulators for developing thinking skills. In *ICALT'17, 17th International Conference on Advanced Learning Technologies*, pages 81–83. IEEE.

Pencheva, E. and Atanasov, I. (2016). Engineering of web services for internet of things applications. *Information Systems Frontiers*, 18(2):277–292.

Peng, L. and Ho, P. (2018). A novel framework of distributed datacenter networks to support intelligent services: Architecture, operation, and solutions. *IEEE Access*, 6:77485–77493.

Saurez, E., Hong, K., Lillethun, D., Ramachandran, U., and Ottenwälder, B. (2016). Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *DEBS'16, 10th ACM International Conference on Distributed and Event-based Systems*, pages 258–269. ACM.

Striewe, M. and Goedicke, M. (2014). Automated assessment of uml activity diagrams. In *ITiCSE'14, Conference on Innovation & Technology in Computer Science Education*, pages 336–336. ACM.

Tranoris, C., Denazis, S., Guardalben, L., Pereira, J., and Sargento, S. (2018). Enabling cyber-physical systems for 5g networking: A case study on the automotive vertical domain. In *SEsCPS'18, 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 37–40. IEEE/ACM.

Vidal, E. J. and Villota, E. R. (2018). Sysml as a tool for requirements traceability in mechatronic design. In *ICMRE'18, 4th International Conference on Mechatronics and Robotics Engineering*, pages 146–152. ACM.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS