IEICE TRANSACTIONS

on Information and Systems

VOL. E101-D NO. 6
JUNE 2018

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

The Pre-Testing for Virtual Robot Development Environment

Hyun Seung SON[†a)], ***Nonmember and*** **R. Young Chul KIM**[††b)], ***Member***

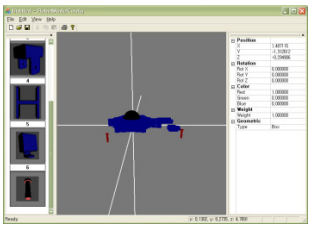
SUMMARY The traditional tests are planned and designed at the early stages, but it is possible to execute test cases after implementing source code. Since there is a time difference between design stage and testing stage, by the time a software design error is found it will be too late. To solve this problem, this paper suggests a virtual pre-testing process. While the virtual pre-testing process can find software and testing errors before the developing stage, it can automatically generate and execute test cases with modeling and simulation (M&S) in a virtual environment. The first part of this method is to create test cases with state transition tree based on state diagram, which include state, transition, instruction pair, and all path coverage. The second part is to model and simulate a virtual target, which then pre-test the target with test cases. In other words, these generated test cases are automatically transformed into the event list. This simultaneously executes test cases to the simulated target within a virtual environment. As a result, it is possible to find the design and test error at the early stages of the development cycle and in turn can reduce development time and cost as much as possible.

key words: *virtual environment, pre-testing, automatic test cases generation, simulation*

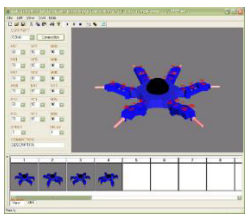
1. Introduction

In the software development life-cycle, the error finding stage has a significant range of expenses depending on which stage the error is found. At the implementation stage it will cost 30∼100 times more than the cost for fixing errors in the requirement stage [1]. Software errors may be found out during testing. If the software test can be executed earlier, then it will reduce the time of development and cost. Yet, still in today's software industries, the conventional tests are used after development is complete [9]. To test at the earlier stage, several papers mentioned the automatic generation method of test cases in the virtual simulation environment [2]–[6].

Based on the previous test case generation method, we suggest a way to automatically generate test cases that provide four coverages for state, transition, instruction pair, and all paths [6]. Our proposed method includes 'pre-modeling & simulation' process along with test case generation. Some papers extended 'pre-modeling & simulation' process for assembling and pre-testing of virtual multi-jointed robot before developing controlled joints [2]–[5].



(a) The model combination tool



(b) The motion generation tool

Fig. 1 The virtual protoyping tool named VirRobot.

Our VirRobot [3]–[5] is a virtual-prototyped multi-jointed robot, which was developed by Hongik University for KMC Robotics Corp. in Korea. This is utilized as a robotic training equipment in many domestic schools. The VirRobot can design a multi-jointed robot consisting up to 10 legs with a camera and multiple sensors to detect ultrasonic waves and temperatures. In addition, the VirRobot provides both model combination and motion generation facilities. The Fig. 1 (a) illustrates the model combination tool generated by VirRobot.

It is possible to determine whether the dynamic robot behavior is correct or not. The test case generation method is to generate test cases based on the state diagram. The test cases are identified with test case IDs, which include four coverages such as the state coverage, transition coverage, instruction pair coverage, and all path coverage, which are all based on the state transition tree. To create test cases with the state transition tree, we adapt MOF (Meta Object Facility) [7] and XMI (XML Metadata Interchange) [8]. However, this method does not necessary to wait until the implementation stage since it can have the executing test cases in the virtual simulation environment even at the design stage. Therefore, the design error can be found in early stage.

This paper is organized as follows. Chapter 2 describes related work. Chapter 3 explains our virtual pre-testing process. Chapter 4 shows a simple case of our proposed method. Finally, Chapter 5 describes the conclusion.

2. Related Work

In model based testing, Andras Toth et al. [10] had proposed the framework for the model level testing of UML model. As of the result of this UML, the design can be tested. The design flow can be found in the modeling stage of development process. This is so that the realization activity to save

Manuscript received August 3, 2017.
Manuscript revised January 5, 2018.
Manuscript publicized March 1, 2018.

[†]The author is with Department of Reliability Technology Institute, Moasoft, Seoul, 05770 Korea.
[††]The author is with Department of Computer and Information Communications, University of Hongik, Sejong, 30016 Korea.
a) E-mail: hson@moasoftware.co.kr
b) E-mail: bob@hongik.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2017EDP7249

significant effort and expenses.

The extraction of model based test case in the formal conformance testing for UMLSC (UML State-Chart) had been suggested by Stefania Gresi et al. [11].

Bertolino A. et al. [12] have integrated the sequence diagram with the state of extracting reference model, perfectly. This is automatically used to extract the test case based on the UML specification. The objective of the paper is processing a perfect model from the state and sequence diagrams for UIT (User Interaction Test). This method guarantees the inclusion of all the allowable sequences. However, it cannot provide any coverage to measure the system. The advantage of this method is to create the accurate test case.

3. Our Virtual Pre-Testing Process

This approach consists of a total of seven stages of the development cycle: requirements, pre-modeling, pre-simulation, design, implementation, model based testing, and code based testing as shown in Fig. 2.

The requirement stage (Step 1) extracts and analyzes requirements for the virtual robot. The pre-modeling stage (Step 2) assembles the parts (motor, sensor, and MCU) of the virtual target object (that is, a robot), and extracts the motion of it. The selected part in this stage is linked with the hardware profile. When assembling the target object, a robot is assembled in the desired form with the parts of hardware components built in M&S. For example, the robot consists of the head, the body, and the legs according to its characteristic and parts. The suitable position has its roles, which connects all joints to give complete movement to the robot. The dynamic robot behavior can be created easily with the 'motion capture method' through extracting serial motions of virtual object. By using motion capture, it can see and save the degree of each angle depending on all dynamic behaviors. The robot can be moved by restoring the sequentially captured & saved data. It may be similar to play the animation.

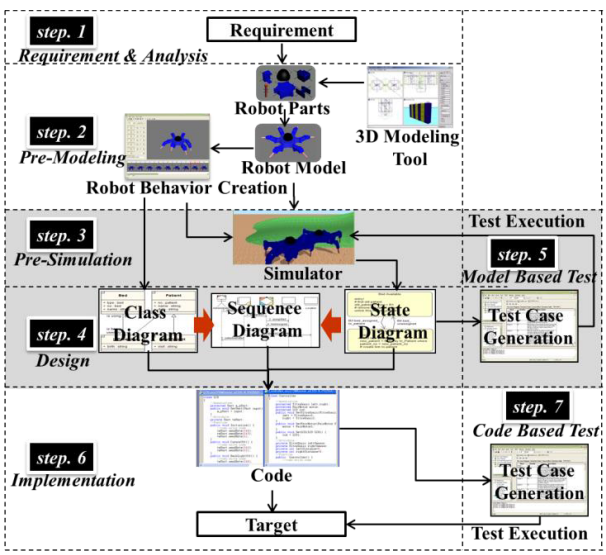


Fig. 2 Model based testing in M&S.

At the pre-simulation stage (Step 3) analyses the robot's environment, and simulates the dynamic robot movements to occur. This is to find the potential problem to move it. In the virtual environment the robot can execute its mission just as the same as it works on in the real environment. Exterior elements in the virtual environment are weather, topography, and objects. That is, it simulates all movements of object in virtual physics engine like completion of robot model and motions. It is also possible to check the dynamic robot behavior in this stage, such as determining whether it may move in incompatible directions.

At the design stage (Step 4), the robot model is constructed with UML. In this stage, the robot is designed using class, sequence, and state diagram. The parts of the robot can be represented with the class of each part. At this time, it can set each part of the robot with its role, attribute, and behavior. At model based testing (Step 5), it can create the test case based on the model generated in the design stage. The target object in the M&S environment is also tested. Based on the state diagram, the state transition tree is generated to create test cases. We can asynchronously test the simulated model in the virtual environment along with test cases generated at the design stage. To execute the pre-test without code in the simulation environment, the simulated robot can be activated with the action language, that is, the intermediate language. Therefore, the robot can move with the action language in the virtual environment without complete production. In other words, it can be tested through black box testing without code in M&S simulation tool.

At the implementation stage (Step 6), it can be possible to generate the code based on the design model. The target code is created with the diagram metamodel and code template. At code based testing (Step 7), the white box test based on the code generated in implementation stage it carried out. The quality can be improved even further by executing both model based testing and code based testing even after complete development.

This paper just focuses on Step 3, Step 4, and Step 5 within the proposed test process.

4. Test Case Generation and Execution

Our approach is focused on model based test case generation & execution, especially the state diagram, which verifies the relationships among the event, behavior, action, state, and state transition. With this technique, it can determine whether the dynamic behavior of the robot (that is, the changing of the state of the robot) can satisfy the system specifications or not.

There are three causes of the fault mechanism of the state based system as follows: the first is that state changing of state diagram cannot transit according to the system function specification accurately. The second is that the syntax of state diagram is wrong or inconsistent. The third is

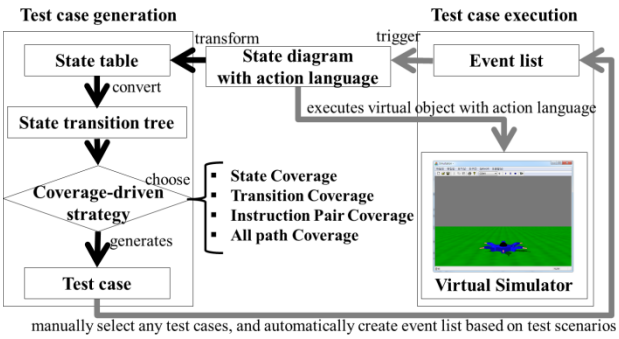


Fig. 3 Testing procedure of the test case generation and execution.

the conversion from state diagram to the test code. In this case, it may cause troubles if converted manually. Therefore, it does not matter if automatically converted with the automatic tool.

For example, Fig. 3 shows the whole testing procedure for test case generation and execution. First in test case generation, the state table is transformed with the state diagram, which is converted into the state transition tree. Then we can generate test case with this transition tree based on coverage-driven strategy. In test case execution, we manually select test cases from the generated test cases with any coverage driven strategy. Then we automatically generate the event list based on test scenarios which consist of Test Case ID, Type, State/Event, and Pass/Fail (P/F). Finally, we execute each event of the whole event list sequentially.

Each event triggers the transition from one state to other state of the state diagram and executes the virtual robot with action language in a virtual simulator. The action language is used to work a virtual object in a virtual simulator without even complete development. As a result, the execution of test cases triggers the state diagram which reflects the virtual simulator to execute the virtual robot. This then confirms the testing result of the simulator. From this we can recognize and decide if it is a pass/fail.

4.1 Conversion of State Diagram to State Transition Tree

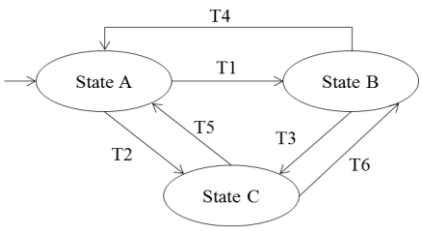
Initially, the state diagram is converted into the state table. The state table has been separated with state and event. This can represent states of all situations. From this, we create the state transition tree based on the state table. State transition tree has all the possible movable states repeatedly. The test case level is varied according to the frequently respective execution.

To convert the state table into the transition tree, we suggest a different algorithm based on Binder's algorithm [16]. Thus, we can easily convert the state table to the transition tree. One of the reasons why we use the state table is because we can correctly identify the missing state, the incorrect transition, or the missing event. Figure 4 shows our algorithm to analyze from a starting state to all states in a stable table, which marks the state as visited.

This generates the transition tree like the breadth first search (BFS), as seen in the generated transition tree with

1. Create the root node of the transition tree from the 'start' state in the state diagram.
2. Identify the 'start' state as the root node in the state table.
3. From the state in step 2, create the child nodes and link edges on state transition tree in the column order like out-branches of the state. With the created states, then mark '√' in 'visited' column of the state table.
4. Visit each node of the child states repeatedly in the column of the top row in identified child nodes of the state until visiting all states.
5. Repeatedly execute step 3 and step 4 until marking '√' in 'visited' column on visiting states in each column of all rows of the table, except the already 'visited' state.

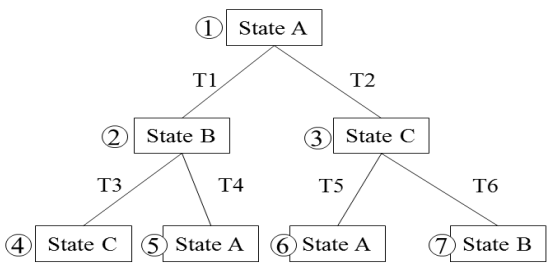
Fig. 4 Our suggested algorithm.



(a) State diagram

State / Event	① State A		State B		State C	
	state	visited	state	visited	state	visited
T1	② State B	√	N/A	x	N/A	x
T2	③ State C	√	N/A	x	N/A	x
T3	N/A	x	④ State C	√	N/A	x
T4	N/A	x	⑤ State A	√	N/A	x
T5	N/A	x	N/A	x	⑥ State A	√
T6	N/A	x	N/A	x	⑦ State B	√

(b) State table



(c) State transition tree

Fig. 5 The conversion process of State Transition Tree from State Diagram.

the numbering order in Fig. 5 (b). We add a 'visited' column in the state table to check whether the state is visited or not.

Figure 5 (a) shows a simple state diagram. Figure 5 (b) shows how to make the state table based on the state diagram. The available state transits toward another state meeting at the event. Then each state can be indicated on the top of table and the event on the left side of the table. For example, the out-branches on State A are T1 and T2. The

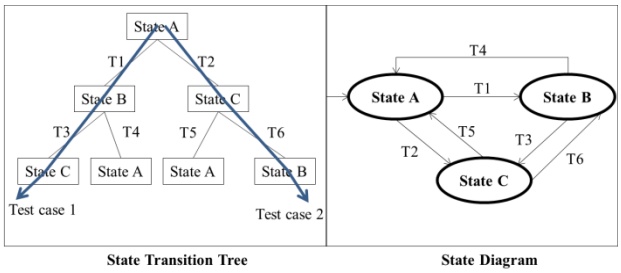


Fig. 6 State coverage of state diagram.

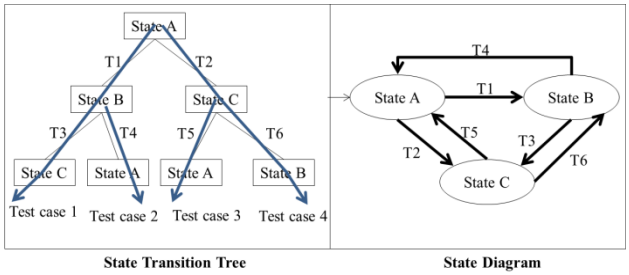


Fig. 7 Transition coverage of state diagram.

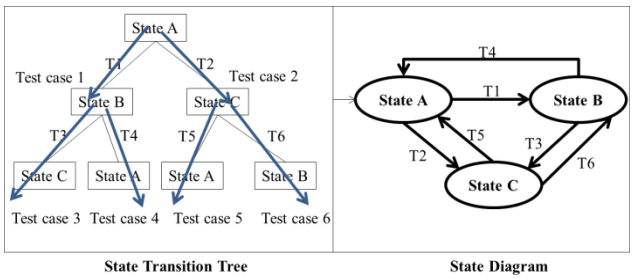


Fig. 8 Instruction pair coverage of state diagram.

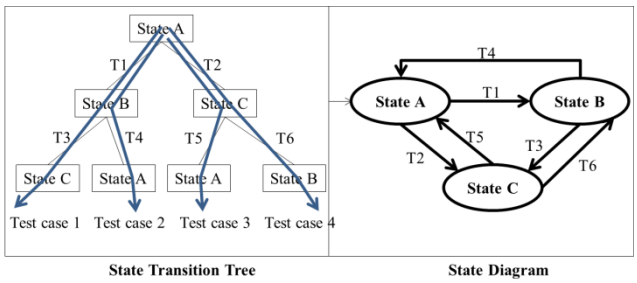


Fig. 9 All path coverage of state diagram.

out-branches on State B are T3 and T4. The out-branches of State C are T5 and T6. It also marks the numbering on each state like Fig. 5 (b).

Figure 5 (c) shows how to create the transition tree. It lists all the states on the top of table in the sequence. Then, it indicates all the available states for access in the next step. We can possibly generate different test cases from the state transition tree according to the choice of any coverage driven strategy. All these possible paths in the sequence are the test cases.

4.2 Coverage-Driven Strategy

The coverage means a complete set of tests which can be measured. It is possible to apply one design model [13]. Therefore, this paper applies coverage from the state transition tree based on the state diagram. This generates test cases with the tree. These coverages have four types, which are state coverage, transition coverage, instruction pair coverage, and all path coverage.

State coverage has a rule that requires a state at least one time to cover all states to execute. We can extract two types of test cases to satisfy 100% of coverage after transforming the state transition tree from the state diagram like Fig. 6. In this case of coverage, two test cases must be satisfied such as test case 1(A-B-C) and test case 2(A-C-B). This coverage is similar to transition coverage. However is should begin from the starting state to the end state over all paths. In this state coverage, to generate test cases to satisfy 100% of this coverage, it becomes to find all paths of states and transitions.

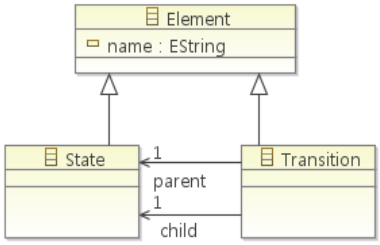
Transition coverage has a rule that requires a transition at least one time to cover all transitions of the applied model to execute. After transforming state transition tree we can extract four types of test cases to satisfy 100% of coverage from state diagram like Fig. 7. In these cases of coverage, it must satisfy four test cases such as test case 1(A-B-C), test case 2(B-A), test case 3(C-A), and test case 4(A-C-B). This coverage is just to visit a transition one time, not necessarily going back the starting state again. To generate test cases to satisfy 100% of the transition coverage, it should find all of the state coverage. Then is should add partial paths of no executed transitions.

Instruction pair coverage has a rule that requires a pair of state and transition at least one time to cover instruction pair coverage in order to execute. After transforming state transition tree from state diagram, we can extract six types of test cases to satisfy 100% of coverage like Fig. 8. In these cases of the coverage, it must satisfy test case 1(A-B), test case 2(A-C), test case 3(B-C), test case 4(B-A), test case 5(C-A), and test case 6(C-B). This coverage consists of a pair of a state and a transition. It should find no overlapped paths with a state and a transition in order to generate test cases to satisfy 100% of coverage.

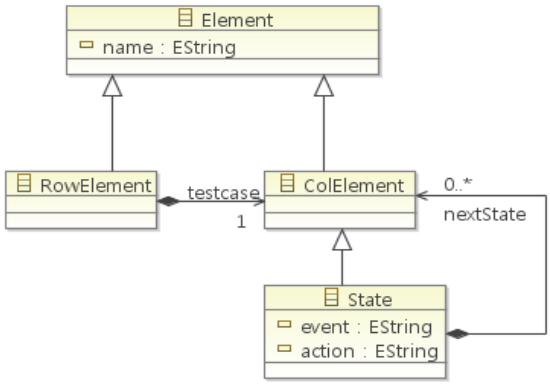
All path coverage has a rule that travels all possible paths of state transition tree. After transforming state transition tree from state diagram, we can extract four types of test cases to satisfy 100% of coverage like Fig. 9. In these cases of coverage, it must satisfy test case 1(A-B-C), test case 2(A-B-A), test case 3(A-C-A), and test case 4(A-C-B). This coverage is similar to transition coverage, but should begin from the starting state over all paths. It should find all possible paths via all states and transitions to generate test cases to satisfy 100% of coverage.

4.3 Test Case Generation with Metamodel Mechanism

To automatically generate test case description from the



(a) Metamodel of state transition tree



(b) Metamodel of test case description

Fig. 10 Metametamodel for test case generation.

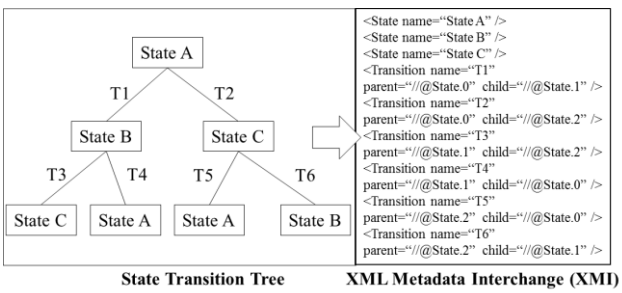


Fig. 11 XMI representation of state transition tree based on metamodel.

state transition tree, we propose this generation with metamodel mechanism. Metamodel is used with MOF (Meta Object Facility) as a language to define a model. It also uses XMI to represent MOF data. Figure 10 shows each metamodel of the state transition tree and test case description.

This state transition tree is represented with corresponding states and transitions. We separate the state element and the transition element. Only the state element is used, and the transition element is linked with the state and the referenced tree. Figure 11 shows the XMI representation of state transition tree based on this metamodel. The states are A, B, C and the transitions are T1~T6. These link states with transitions based on attributes of parent and child states.

Test case description consists of two parts, *RowElement* and *ColElement*. The *RowElement* identifies the serial order of test cases. The *ColElement* represents the process of the transition between states.

In the *ColElement*, we are able to represent the path of state and transition in order to link states continuously

TCID	Initial State	Event	Action	Next State	Event	Action	End
TC1	S_A	E1	Do	S_B	E2	Do	S_A
TC2	S_B	E2	Do	S_A	E1	Do	S_B

```
<RowElement name="TC1">
  <testcase name="TC1">
    <State name="S_A" event="E1" Action="Do">
      <nextState name="S_B" event="E2" Action="Do">
        <nextState name="S_A" />
      </nextState>
    </State>
  </testcase>
</RowElement>
<RowElement name="TC2">
  <testcase name="TC2">
    <State name="S_B" event="E2" Action="Do">
      <nextState name="S_A" event="E1" Action="Do">
        <nextState name="S_B" />
      </nextState>
    </State>
  </testcase>
</RowElement>
```

Fig. 12 XMI representation of test case description based on metamodel.

```
for(int i=1; STT.Testcases; i++) {
        RowElement rowE = Create(RowElement);
        rowE.name = "TC"+i;
        STT.Transition transitions = STT.Testcases[i-1];
        State firstState = null;
        for(int ti=0; transitions; ti++) {
                if(ti == 0) { //first
                        firstState = Create(State);
                        firstState.name = transitions[ti].parent.name;
                        firstState.event = transitions[ti].name;
                        firstState.action = transitions[ti].parent.acttion;
                        rowE.testcase = firstState;
                }
                else if(ti == transitions.length-1) { //last
                        State endState = Create(State);
                        endState.name = transitions[ti].parent.name;
                        firstState.nextState.add(endState);
                }
                else {
                        State state = Create(State);
                        state.name = transitions[ti].parent.name;
                        state.event = transitions[ti].name;
                        state.action = transitions[ti].parent.acttion;
                        firstState.nextState.add(state);
                }
        }
}
```

Fig. 13 A transformation algorithm for test case.

longer. Figure 12 represents the XMI test case description. Each test case ID is represented on *RowElment*. With the information of *RowElment*, each test case ID represents a links state and transition.

We can transform two different types of data with this metamodel from the State Transition Tree. Figure 13 shows the transformation algorithm for test case creation from the state transition tree with the metamodel. This algorithm is described as follows. First, all possible test cases are extracted. Then, the test case ID is produced and information within *RowElements* is filled. Secondly, a data is developed sequentially (initial state, event, action) with the test case. It repeatedly executes all states and transitions within the test case.

Finally, the test cases with the state transition tree is generated. Generated test cases are shown in the Table 1. Generated test cases describe the scenario executed by states within the paths of the state transition tree.

Table 1 Generated test case.

ID	Initial State	Action	Event	Next State	Action	Event	End
TC1	S_A	Do	E1	S_B	Do	E2	S_A
TC2	S_B	Do	E2	S_A	Do	E1	S_B

4.4 Test Case Execution in a Virtual Environment

It is possible to model the change of the state with only a state diagram. However, it is impossible to control the robot in a virtual environment because of describing the behavioral movements like black-box style. To solve this problem, we suggest using a state diagram with action language. Action language is a set of demanded commands to control the robot in a virtual environment. Writing the commands with the action language, described in the state or transition of the state diagram, is advised. After one state is transited, the described commands are sent with the action language in the state and into the simulator. This is to control the robot in a virtual environment.

These commands have two types: state control and motion control. In the motion control commands, there are five commands to execute '*forward*', '*backward*', '*leftturn*', '*rightturn*', *and* '*stop*' such as moveForward(), moveBackward(), moveLeftTurn(), moveRightTurn(), moveStop() like Fig. 14.

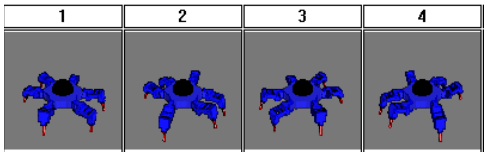
Figure 15 shows the state diagram with Action language. It represents the forward movement until the condition 'time' <= 10 in 'Forward' state, and to go to the 'Stop' state if 'time' > 10.

In the state control commands, there are six commands to control the robot with the data from the front sensor, the rear sensor, the left sensor, and the right sensor. These include getFrontSensor(), getRearSensor(), getLeftSensor(), getRightSensor(), getLocationX(), getLocationY() like Fig. 16.

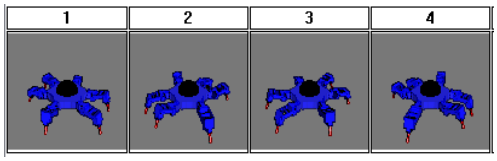
Figure 17 shows the state diagram with Action language. For example, it represents the forward movement until the front sensing value 'fs' >= 10 in 'Forward' state, and go to the 'Backward' state if 'fs' < 10.

Although we use action language on state to execute, it cannot transit other states without the happening of an event. To solve this problem, we make the event list to automatically occur from one event to another. This event list causes it to occur sequentially after each event in state diagram is transiting from a state to the other state. This can possibly help to execute test scenarios linked with test cases. In other words, the generated test case cannot be processed right away in the simulator. Thus to execute, it must convert the test case to the event list as shown in Table 2.

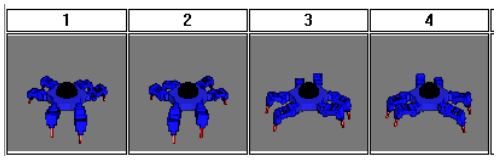
Table 2 is the conversion result of the test cases generated in the Table 1 to the event list. TCID is the ID of test case. In the Type column, S represents a state, and E represents an event. State/Event is assigned in plural so both state and event can come in. Test data means input value to execute test, and also voluntarily input the value of guard



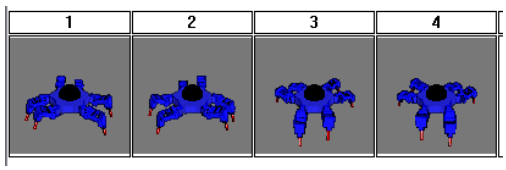
(a) Action command: moveForward()



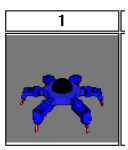
(b) Action command: moveBackward()



(c) Action command: moveLeftTurn()



(d) Action command: moveRightTurn()



(e) action command: moveStop()

Fig. 14 Execution result of action command over action language.

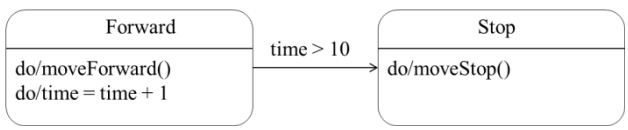


Fig. 15 An example of motion control of action language.

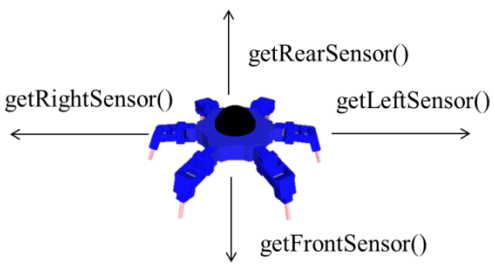


Fig. 16 Execution result of status command over action language.

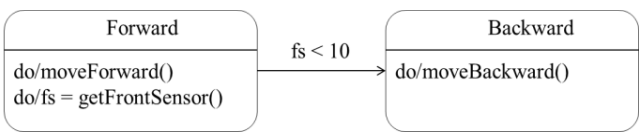


Fig. 17 An example of status control of action language.

Table 2 Event list.

Test Case ID	Type	State/Event	Test data	P/F
TC1	s	S_A		
TC1	e	E1	N/A	
TC1	s	S_B		
TC1	e	E2	N/A	
TC1	s	S_A		
TC2	s	S_B		
TC2	e	E2	N/A	
TC2	s	S_A		
TC2	e	E1	N/A	
TC2	s	S_B		

condition on a transition at changing among states of state diagram. In Table 2, there does not exist the guard condition on an event provided it represents 'N/A' notation of each test data. It also finds the examples in Table 4.

P/F means Pass/Fail. The state and event in the event list are carried out alternatively, and the robot executes the behavior whenever the event list is processed in the virtual simulator environment.

5. Case Study

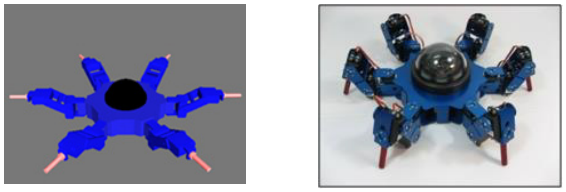
Figure 18 (b) shows a multi-joint robot with six arms. 8 bits-Atmega 128 and programming C language are installed. It uses 18 motors to control multi-joints built by KMC Robotics Inc., Korea [14], [15]. This robot has ultrasonic wave sensors to sense an obstacle on front/rear/left/right of it. We simulate this articulated robot in a virtual environment like Fig. 18 (b).

This robot moves from the starting point to the destination (the goal point) with route scenarios like Fig. 19. The sensors aid the robot to move and to avoid an obstacle from the starting point to the goal point. In order to avoid the obstacle, we assign the minimum boundary value. If the sensing value is greater than the minimum, the robot continuously moves. Otherwise, the robot rotates. It repeatedly executes this process until arriving the goal point.

-Modeling a state diagram:

The application case shows the articulated robot moving to the target position with four directions: forward, backward, left turn, and right turn. Actually, this six arms robot cannot walk straight on forward or backward movement. Therefore, we should consider a revision in the forward and backward movement on modeling the state diagram. In this case, we design to walk this robot not to encounter obstacles against the right wall. In order to model the possible changing states of the robot, we show the state diagram in Fig. 20. To find a design error of the state diagram, we insert ①②③ errors of the wrong transitions into the state diagram. Figure 20 shows the state diagram with the design errors.

Table 3 shows the constant values for sensor of the state



(a) Virtual robot (b) Physical robot

Fig. 18 Articulated robots in virtual and physical environment.

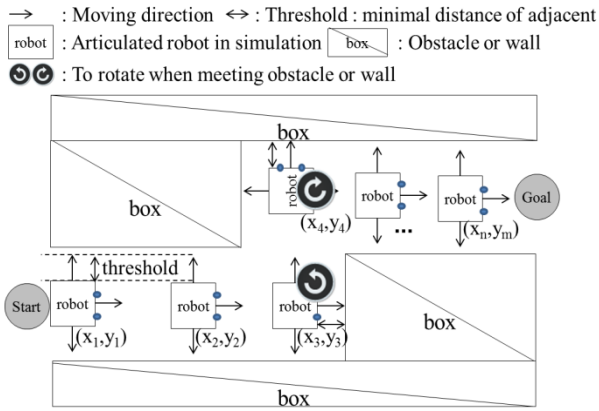


Fig. 19 The behavior scenario of articulated robot.

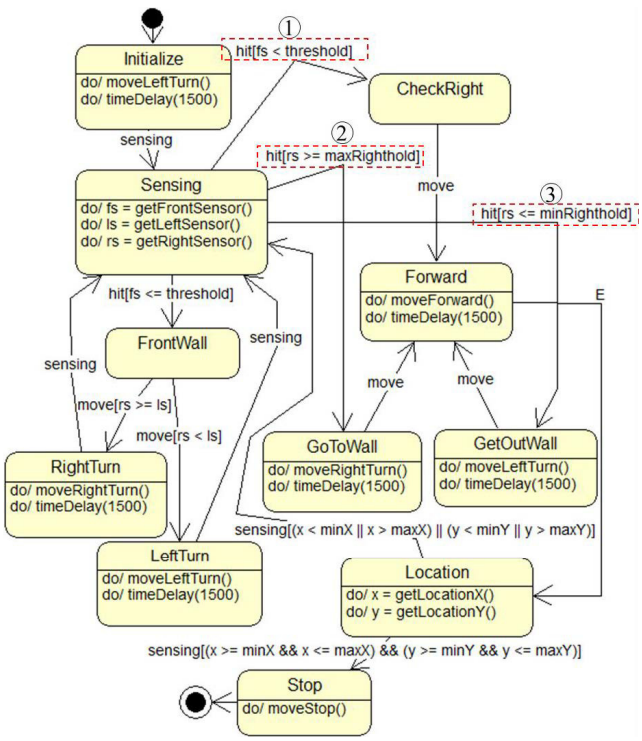


Fig. 20 A state diagram with design errors of articulated robot.

diagram.

-State table generation:

When a specific event is generated in the corresponding

state, state table draws the called state. As shown Table 4, it is easy to check out which state is called when a certain event is generated in the current state. This information is used to convert state table to state transition tree when creating test cases. We will additionally use check modeling which state or transition is missing in state diagram.

- State transition tree generation:

State transition tree as shown Fig. 21 is generated based on the current state table. State transition tree draws all the states in the form of tree. This can be called in the current state for the visual checking. The tree will gain more branches as the depth assigned by selecting Switch.

Table 3 Constant values for simulation.

Name	*Value*	*Name*	*Value*
threshold	5	minX	12
sidehold	3.5	minY	3
maxRighthold	7	maxX	14
minRighthold	5	maxY	7

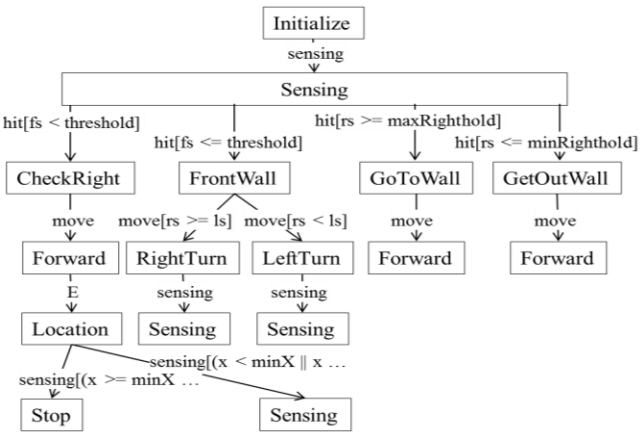


Fig. 21 A generated state transition tree from state table.

- Test case generation:

To generate test case from state transition tree, we use the transition coverage of the coverage-driven strategy. Table 5 shows the generated test cases.

Test case indicates that event and action can be occurred in the current state (Start state). It can also come up in a form of a table through the current state, which will

Table 5 A generated test case.

ID	*Initial State/ Next State*	*Action*	*Event*	*Next State/ End*
TC1	Initialize	do/moveLeftTurn() do/timeDelay(1500)	sensing	Sensing
〃	Sensing	fs=getFrontSensor() ls=getLeftWensor() rs=getRightSensor()	hit [fs<threshold]	CheckRight
〃	CheckRight	N/A	move	Forward
〃	Forward	do/moveForward() do/timeDelay(1500)	E	Location
〃	Location	do/x=getLocationX() do/y=getLocationY()	sensing [x>=minx…	Stop
TC2	Location	do/x=getLocationX() do/y=getLocationY()	sensing [x<minx ‖ x…	Sensing
TC3	Sensing	fs=getFrontSensor() ls=getLeftWensor() rs=getRightSensor()	hit [fs<=threshold]	FrontWall
〃	FrontWall	N/A	move[rs>=ls]	RightTurn
〃	RightTurn	do/moveRightTurn() do/timeDelay(1500)	sensing	Sensing
TC4	FrontWall	N/A	move[rs<ls]	LeftTurn
〃	LeftTurn	do/moveLeftTurn() do/timeDelay(1500)	sensing	Sensing
TC5	Sensing	fs=getFrontSensor() ls=getLeftWensor() rs=getRightSensor()	hit[rs >= maxRighthold]	GoToWall
〃	GoToWall	do/moveRightTurn() do/timeDelay(1500)	move	Forward
TC6	Sensing	fs=getFrontSensor() ls=getLeftWensor() rs=getRightSensor()	hit[rs <= minRighthold]	GetOutWall
〃	GetOutWall	do/moveLeftTurn() do/timeDelay(1500)	move	Forward

Table 4 A generated state table.

	Initialize	*Sensing*	*FrontWall*	*LeftTurn*	*RightTurn*	*CheckRight*	*Forward*	*GoToWall*	*GetOutWall*	*Location*	*Stop*
hit[fs < threshold]	N/A	CheckRight	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
E	N/A	N/A	N/A	N/A	N/A	N/A	Location	N/A	N/A	N/A	N/A
sensing[(x >= minX…	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Stop	N/A
sensing[(x < minX ‖ x…	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Sensing	N/A
sensing	Sensing	N/A	N/A	Sensing	Sensing	N/A	N/A	N/A	N/A	N/A	N/A
hit[rs >= maxRighthold]	N/A	GoToWall	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
hit[rs <= minRighthold]	N/A	GetOutWall	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
move	N/A	N/A	N/A	N/A	N/A	Forward	N/A	Forward	Forward	N/A	N/A
hit[fs <= threshold]	N/A	FrontWall	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
move[rs < ls]	N/A	N/A	LeftTurn	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
move[rs >= ls]	N/A	N/A	RightTurn	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

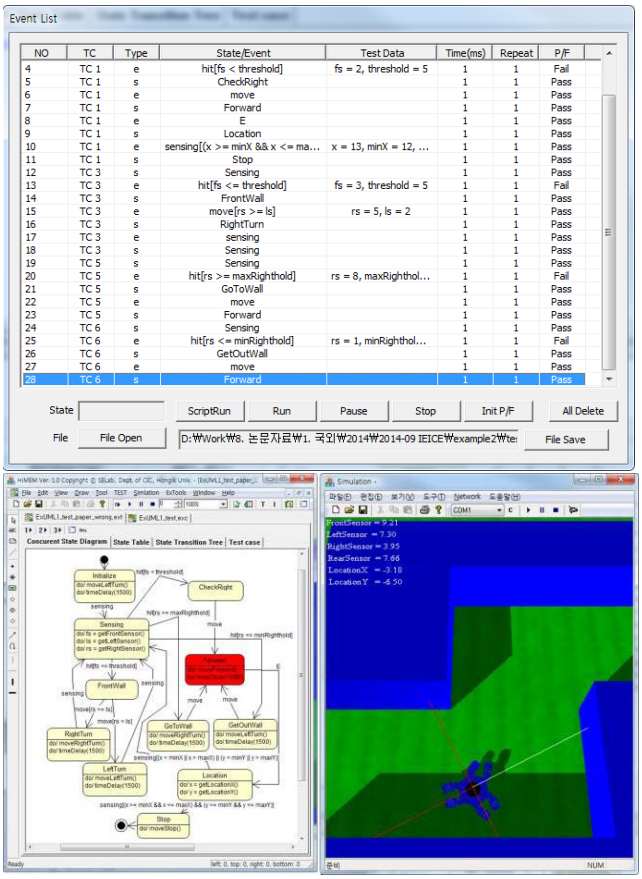


Fig. 22 The result of test case execution.

transmit the next state.

- Generation of event list:

These test cases are inserted in the event list. Test case (TC) mentions test case ID. This consists of the basic unit such as number, TC, type, state/event, repeat, and P/F.

- Running event list:

An event list is produced in the previous step into state diagram engine when loading. The engine executes test cases and simultaneously sends the commands to the simulator. From there, we can identify the design error of the state diagram. This is because we can see the right/wrong movements of the robot with each test scenarios in the simulator as shown Fig. 22.

On asynchronously executing the event list of test cases (in Table 6) with the simulation tool, we can detect four errors shown as Fig. 22. According to the test data, our expected value is true against the guard condition of an event. However, the actual executed result comes out as a failure. That is, there is a wrong value of event transited from the sensing state to other state.

In Table 6, the ① case assigns a wrong sign of inequality about the 'hit [fs < threshold]' guard condition of

Table 6 A selected event list.

	TC ID	*Type*	*State/Event*	*Test data*	*P/F*
	TC1	s	Initialize		P
	TC1	e	Sensing	N/A	P
	TC1	s	Sensing		P
①	TC1	e	hit [fs<threshold]	fs=2,threshold=5	F
	TC1	s	CheckRight		P
	TC1	e	Move	N/A	P
	TC1	s	Forward		P
	TC1	e	E	N/A	P
	TC1	s	Location		P
	TC1	e	sensing [x>=minx…	x=13, y=5, minX=12,maxX=14, minY=3,maxY=7	P
	TC1	s	Stop		P
	TC3	s	Sensing		P
②	TC3	e	hit [fs<=threshold]	fs=3,threshold=5	F
	TC3	s	FrontWall		P
	TC3	e	move[rs>=ls]	rs=5, ls=2	P
	TC3	s	RightTurn		P
	TC3	e	sensing	N/A	P
	TC3	s	Sensing		P
	TC5	s	Sensing		P
③	TC5	e	hit[rs >= maxRighthold]	rs=8,maxRighthold=7	F
	TC5	s	GoToWall		P
	TC5	e	move	N/A	P
	TC5	s	Forward		P
	TC6	s	Sensing		P
④	TC6	e	hit[rs <= minRighthold]	rs=1,minRighthold=5	F
	TC6	s	GetOutWall		P
	TC6	e	move	N/A	P
	TC6	s	Forward		P

state/event. The ② case also assigns a wrong sign of less and equality about the 'hit [fs <= threshold]' guard condition of the state/event. They make wrong executions because these cases have assigned with the same 'less' signs. Therefore, to make the right design of a state diagram, the 'hit [fs < threshold]' guard condition must be changed with 'hit [fs > threshold]'. In the 'hit[rs >= maxRighthold]' and 'hit[rs <= minRighthold]' guard conditions in the '*Sensing*' state in Fig. 20, the value of 'rs' variable determines to transit some states ('*GoTOWall*' state *or* '*GetOutWall*' state). Yet, the value of the 'fs' variable also determines to transit other states ('*FrontWall*' state or '*CheckRight*' state). When an event comes in a sensing state, it causes to transit two different states in this state diagram, which are not even concurrent model. So, it turns out to be a failure because of impossibly transiting both states. Therefore, these two 'hit[rs >= maxRighthold]' and 'hit[rs <= minRighthold]' guard conditions should be changed to other state. Throughout the process of correcting errors, it is possible to make the right design of state diagram in Fig. 23.

Concurrently, by running the modified state diagram

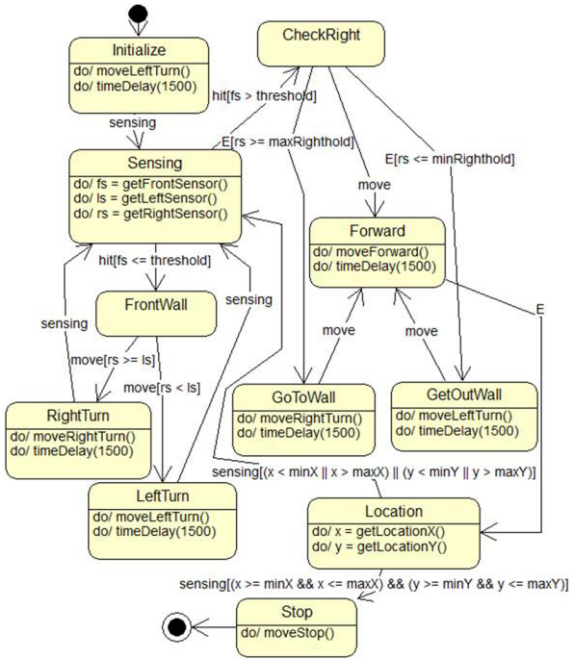


Fig. 23 The fixed state diagram of articulated robot.

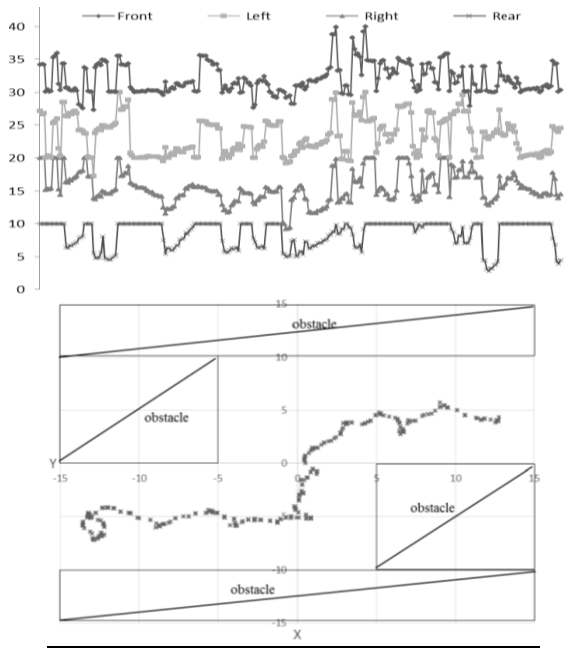


Fig. 24 The results of sensor data of the fixed state diagram performed in simulator.

(in Fig. 23) with the simulator of the right side (in Fig. 22), the virtual robot can reach from the starting point to the goal point like the behavior scenarios of the robot in Fig. 19. This shows the data values such as the front, rear, left, right sensor, and the current position (x,y) of the virtual robot during simulating in the virtual environment.

Even though the robot sometimes may move around circle, finally he can reach the destination through avoiding obstacles.

6. Discussion

In this paper, we propose the test case design method for virtual pre-testing in a simulation environment. The goal of our proposed method is focused on verifying the design whether it is corrected or not. But model checking is an automated method, which be done only by a developer who learned this method. In order to apply a model checking to our robot cases, the scenario specification should be transformed into property descriptions in modal logic formula, which is a difficult point of model checking.

On the other hand, our visual approach can easily be found any design problem by every developer. As a result, it is possible to find the design and test error on modeling & simulation (M&S) at the early stages of the development cycle. With our approach, it expects to reduce cost and time of development.

7. Conclusion

Our suggested method consists of test process and test case creation method, which executes testing with pre-modeling and pre-simulation at the design stage. This is different from the previous methods. We adapt metamodel mechanism to automatically generate test cases from a design model, that is, the state diagram. As the result, even in the design stage we can asynchronously execute test cases with the virtual robot in the simulation environment *before complete development*. Therefore our proposed method overcomes the conventional problem, which can find out design errors by executing the test without waiting until its implementation stage. A problem still exists in the proposed method. That is, the tester should visually check, and confirm the test case executed in the virtual environment. Thus, in order to overcome such weakness, we should consider automatic checkup method in future research.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A3B03035421) and the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (NRF-2015H1C1A1035548).

References

[1] B.W. Boehm, Software Engineering Economics, Prentice-Hall, 1981.
[2] H.S. Son, W.Y. Kim, and R.Y.C. Kim, “Implementation of Technique for Movement Control of Multi-Joint Robot,” The 30th KIPS Fall conference 2008, vol.15, no.2, pp.593–596, 2008.
[3] W.Y. Kim, H.S. Son, R.Y.C. Kim, and C.R. Carlson, “MDD based CASE Tool for Modeling Heterogeneous Multi-Jointed Robots,”

CSIE 2009 (IEEE Computer Society), Los Angeles/Anaheim, USA, vol.7, pp.775–779, 2009.
[4] J.S. Kim, H.S. Son, W.-Y. Kim, and R.Y.C. Kim, "A Study on Education Software for Controlling of Multi-Joint Robot," Journal of the Korean Association of Information Education, vol.12, no.4, pp.469–476, 2008.
[5] J.S. Kim, H.S. Son, W.-Y. Kim, and R.Y.C. Kim, "A Study on M&S Environment for Designing The Autonomous Reconnaissance Ground Robot," Journal of the Korea Institute of Military Science and Technology, vol.11, no.6, pp.127–134, 2008.
[6] W.Y. Kim, H.S. Son, and R.Y.C. Kim, "A Study on Test Case Generation Based on State Diagram in Modeling and Simulation Environment," Communications in Computer and Information Science, vol.199, pp.298–305, Springer, 2011.
[7] OMG, "OMG Meta Object Facility (MOF) Core Specification," v2.4.2, OMG Available Specification, 2014.
[8] OMG, "MOF 2.0/XMI Mapping, v2.1.1," OMG Available Specification, 2007.
[9] Ilene Burnstein, Practical Software Testing, Springer-Verlag, 2003.
[10] A. Toth, D. Varro, and A. Pataricca, "Model Level Automatic Test Generation for UML State-Charts," Sixth IEEE workshop on Design and Diagnostics of Electronic Circuits and System, DDECS 2003, 2003.
[11] S. Gresi, D. Latella, and M. Massink, "Formal Test-Case Generation for UML Statecharts," Ninth IEEE International Conference on Engineering Complex computer system Navigating complexity in e-Engineering Age, 2004.
[12] A. Bertolino, E. Marchetti, and H. Muccini, "Introducing a reasonably complete and coherent approach for model-based testing," Electronic Notes in Theoretical Computer Science, vol.116, pp.85–97, 2005.
[13] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, "A study in coverage-driven test generation," Design Automation Conference, Proceedings 36th, pp.970–975, 1999.
[14] R.B. McGhee and A.A. Frank, "On the Stability Properties of Quadruped Creeping Gaits," Mathematical Biosciences, vol.3, pp.331–351, 1968.
[15] M.H. Raibert, "Legged Robots," ACM, vol.29, no.6, pp.499–514, 1986.
[16] R. Binder, Testing Object-Oriented Systems. Models, Patterns, and Tools, Chapter 7 State Machines, pp.175–268, Addison-Wesley, 2000.

Hyun Seung Son received the B.S., M.S., and Ph.D. degree in Software Engineering from Hongik University, Korea in 1999 ~ 2015. He was researcher in Mechatronics Research Center of Hongik University, Korea until 2017. He is currently working on Senior Researcher in Reliability Technology Institute of Moasoft. His research interests are in the areas of Automation Tool Development in Embedded Software, Software Visualization, Metamodel design, and Model Transformation, Model Verification & Validation Method.

R. Young Chul Kim received the B.S. degree in Computer Science from Hongik University, Korea in 1985, and the Ph.D. degree in Software Engineering from the department of Computer Science, Illinois Institute of Technology (IIT), USA in 2000. He is currently a professor in Hongik University. His research interests are in the areas of Test Maturity Model, Model Based Testing, Metamodeling, Software Process Model, Software Visualization.