International Journal of Automotive Technology, Vol. 19, No. 1, pp. 107−119 (2018)
DOI 10.1007/s12239−018−0011−6

Copyright © 2018 KSAE/ 100−11
pISSN 1229−9138/ eISSN 1976−3832

MODEL-BASED AUTOMATIC TEST CASE GENERATION FOR AUTOMOTIVE EMBEDDED SOFTWARE TESTING

Ki-Wook Shin and Dong-Jin Lim*

Department of Electronic Systems Engineering, Hanyang University, Gyeonggi 15588, Korea

(Received 4 October 2016; Revised 10 April 2017; Accepted 23 June 2017)

ABSTRACT−We propose a method to automatically generate software and hardware test cases from a UML model developed through a model-based development process. Where languages such as source-code languages are used within the model, input and expected values for each test case are generated using a custom parser. As a next step, unit test cases are combined to generate integration test cases using a bottom-up approach. Then these cases are converted into hardware test cases for approval testing of embedded systems, using XQuery and hardware mapping tables. We demonstrate this process by applying it to the power window switch module of a Hyundai Santa Fe vehicle. Our approach provides an automatic testing procedure for embedded systems developed by model-based methods, and generates test cases efficiently using a recombination of signals. In conclusion, our proposed method could help reduce the resources needed for test case generation from software to hardware.

KEY WORDS : Software testing, Test case generation, Model-based development, Unified Modeling Language (UML), Integration testing, Power window switch module, Approval testing, Hardware testing

1. INTRODUCTION

In general, a model-based approach to the development of products can lead to greater productivity and more stringent quality assurance through the early detection of design errors in the overall system design process. This applies, in particular, to fields requiring high levels of reliability, such as the automotive industry, where any defects in a product could lead to injury or death. This makes modeling very important to the process of fabrication.

However, much more critical than general software testing in such cases is embedded software testing, due to the complexity of the products being developed and the fact that any system failure could be catastrophic. The process of embedded testing includes planning the test, generating test cases, building a test environment, executing tests, evaluating results, reporting the results, and tracking defects (Bertolino, 2001). This process accounts for roughly 40 % of the overall development cost of a product (Zelkowitz, 1978). To reduce these costs, automation of this process is vital.

Because test case generation is the most important step in the overall testing process, a great deal of research has been conducted on this topic. Ideally, software should be tested for all possible combinations of input values; however, this is difficult to do because of limited time and resources and because the cost of software testing and the time and effort required to manage such tests continue to increase (Tung and Aldiwan, 2000). Furthermore, as software increases in complexity, it is almost impossible to perform tests for all possible conditions. Therefore, efficient, automated test case generation could be key to addressing these problems.

In most studies on automatic test case generation, test cases are generated from sources such as requirement specifications, software models, or source codes. Depending on the sources from which test cases are generated, an abstract test case or actual test case can be generated. There exist many test case generation tools such as AGEDIS and GOTCHA which generate abstract test cases based on models. Abstract test cases cannot be used as they are; instead, they have to be converted to runnable test cases using tools such as TCBeans (Shafique and Labiche, 2010). This approach is well-suited to a model-based testing environment, however, it is hard to select the best test tool for a given development environment, because a specific test tool for interpreting abstract test cases has to be used. In this study, a new method is proposed to generate test cases through parsing of a combined model, which contains not only a platform independent model but also a platform dependent model for an executable model implemented using a commercial UML tool. Also, generated actual test cases are utilized for functional testing using a commercial dynamic testing tool.

Typically, static and dynamic testing play very important roles in the overall testing of software. They help to ensure

**Corresponding author.* e-mail: limdj@hanyang.ac.kr

the quality of the software by identifying bugs and defects that might otherwise be missed (Beizer, 2003). Moreover, embedded software has to be tested on real hardware to verify its performance and operation. Therefore, hardware-in-the-loop (HIL) testing is needed to ensure that the program is operating correctly.

Hardware test cases require as much effort to generate as software test cases. For example, it requires real-time testing equipment, and various parameters (such as the duration of input delays using real-time test equipment) have to be considered. Yet many recent studies on the automatic generation of test cases have focused mainly on software testing rather than hardware testing, with many of the studies manually generating test cases for approval testing. The work necessary to generate hardware test cases can be reduced to a great extent by using the proposed method, which enables automatic hardware test case generation.

Typically, program source code or models can be used to automatically generate test cases. Since model-based test case generation can be used at early stages of development, it is often more effective than source code-based generation (Samuel *et al*., 2008). Prior studies have used UML, Simulink, etc., for model-based test case generation (Offutt and Abdurazik, 1999; Zhan and Clark, 2005), which can be represented using various modeling notations such as scenario-based, state-based, and process-based notation (Anand *et al*., 2013). Specifically, this paper is focused on a state-based UML model.

Model-based testing is one of the most important testing methods in modern software development. A model can be used for simulation during early stages of development, and the developer can refine the software design after finding errors (Bringmann and Kramer, 2008). Additionally, these models contain information that can be utilized to generate test cases for software verification.

In the present paper, we propose a method to automatically generate unit test cases from a Unified Modeling Language (UML) model using custom parser. Based on the generation of custom parsers, integration test cases and hardware test cases can be derived from each of previous test case generation step.

The remainder of this paper is organized as follows. In Section 2, we provide a review of the pertinent literature. Section 3 proposes the new method, and Section 4 applies it to the power window switch module of a Hyundai Santa Fe vehicle. Section 5 summarizes the results and conclusions.

2. Related Work

There has been a great deal of research into the automatic generation of test cases using UML-based models. Samuel *et al*. (2007, 2008) proposed a method for generating test cases from UML state diagrams and communication diagrams. Gulia and Chillar also generated test cases from a UML state diagram using a genetic algorithm (Gulia and Chillar, 2012). Hartmann *et al*. (2000) devised a conversion from several state diagrams into normalized state diagrams and then created integration test cases from the normalized charts. Florin Pinte and Norbert (2008) researched the optimization of integration test cases using genetic algorithms. Ogata and Matsuura (2010) proposed the creation of integration test cases that included input and could predict results from activity, class, and object diagrams. From a higher-level perspective, Heumann (2001) investigated automatic requirement-based test case generation using use case diagrams; however, it was difficult to create the concrete test cases needed for high-level design.

In these studies, test cases were generated from UML models, but platform-independent models (PIM), or requirement-based models, may be insufficient to generate actual test cases. In adopting model-driven engineering technology, abstract test cases are generated from these requirements or PIM, and executed by interpreting infrastructures. However, a selection of specific test tools for interpreting abstract test case could be a constraint on effective test environment development. For example, in automotive industry, software test tools have to contain a tool qualification report required by ISO26262. Due to this problem, commercial tools are used generally, but they do not fully support an interpretation of abstract test cases. Therefore, further studies are required to create actual test cases through conversion of abstract test case or generation from models.

In order to generate actual test cases, recent studies have utilized commercial tools to generate test cases, but these only work if the UML model is described using specific rules. This is possible if the UML model is described using C/C++, Java, or object constraint language (OCL). Nevertheless, an automatic test generator should be able to analyze any kind of description of UML models applied in various development environments. To this end, we propose a new method to analyze UML model descriptions using a custom parser. In earlier studies in which test cases were generated using custom parsers, test cases were generated from source code or platform-independent models (Fraser and Wotawa, 2007). As stated above, to apply this technique in the automotive industry, actual test cases are generated from a model directly instead of converting abstract test cases to actual test cases.

For generating test cases various approaches are used, including search-based test case generation and divide-and-conquer (Chen *et al*., 2012). To generate test cases using a model, genetic algorithms can be used to find test cases (Lefticaru and Ipate, 2007). With respect to non-model-based automatic test case generation, Windisch studied software test case generation for structural code coverage using particle swarm optimization (PSO). PSO performed better than genetic algorithms (GA) for some industrial problems (Windisch *et al*., 2007). In this paper, both algorithms are included: the divide-and-conquer approach is proposed for generating integration test cases, and simulated annealing is applied to find near-optimal solutions.

Hartig *et al*. (2009) introduced a new method of converting software test cases into hardware test cases using model-in-the-loop (MIL) testing. Then these were converted into appropriate hardware test cases for hardware-in-the-loop testing. The logical inputs and outputs of software test cases were mapped to hardware signals. The advantage of this method is that it is unnecessary to recreate test cases for use in acceptance testing. In this paper, a generated integration test case is converted into a hardware test case using a hardware information table, and used for HIL testing.

3. TEST CASE GENERATION

Model-based development techniques can find errors at an early stage in the system design process and thus reduce errors in the end product. UML is normally used for such model-based development and could become the standard method for visualizing software design. A model-based development approach can be applied step-by-step during the testing process, depending on the model used.

In the typical case of embedded software, unit testing should be done to verify the individual software functions. In addition, integration testing is a necessary method to verify the overall software functions integrated for each unit. Finally, hardware testing is required to ensure all functions operate correctly, even when the software is downloaded on to the actual hardware. Therefore, suitable MIL, software-in-the-loop (SIL), and HIL tests must be performed as part of the regression testing of model-based software. This process requires test cases for each stage. However, most studies on the automatic generation of test cases have been limited to the software level.

This paper proposes a new method for the generation of test cases, using a custom parser from a UML model, which can be applied to embedded software as shown in Figure 1. First, unit test cases are generated via a UML

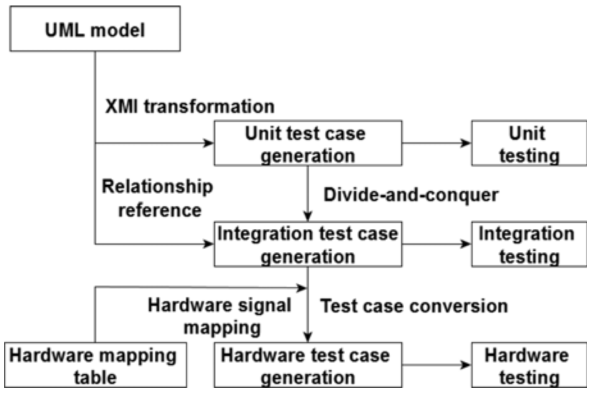


Figure 1. Process for automatically generating model-based test cases.

state diagram based on a breadth-first search and then these are used for unit testing. During this process, to deal with the various types of UML specification language, a new software structure of test case generator is proposed based on a custom parser generator. In the second step, integration test cases are generated from the previously generated unit test cases and the relational diagrams in the UML model. In the final step, hardware test cases are converted from the integration test cases using a hardware mapping table, and then these are used to test the software on embedded systems.

3.1. Unit Test Case Generation

First, UML model metadata are extracted from the model using the Extensible Markup Language (XML) metadata interchange (XMI), which allows the information to be exchanged with external programs.

If software is developed via a model-based approach, the model will contain a great deal of software information. This is why our method uses metadata from the UML-based modeling tool using the XML metadata interchange.

In UML modeling, the behaviors of a system can be represented by behavioral diagrams. The software specifications are normally modeled using the UML standard. However, it is sometimes necessary to describe platform-dependent specifications using any description language such as source-level languages. In this paper, to generate test cases from state diagrams represented by various methods, we propose a process for analyzing each specification (Figure 2).

Considering the example shown in Figure 2, suppose that a transition in UML is descripted using the C language, not an action language. Then it is necessary to parse the text by a parser, which is generated by a parser generator with a grammar for C language. Because parser generators such as ANTLR can generate a lexer and parser from a grammar file, the statement can be parsed by a grammar, which is suitable for ANSI C. Parsed text can be represented in a tree structure such as an abstract syntax tree. Then this is used to generate the test cases. One of the advantages of this method is that it is possible to parse state diagrams through a custom parser regardless of the programming language used.

For example, to describe a functionality of system from a model based development software, a natural language or a program language can be used depending on the model abstract level. If a platform independent model will be made for architecture design, it needs to be considered that the model should not be implemented using a specific program language as possible. After then, when a platform dependent model is made based on the platform independent model, a specific program language such as C/C++, C# and Java can be used to implement a specific behavioral model. These specific behavioral models can be helpful in simulation of the software model and automatic generation of source code.

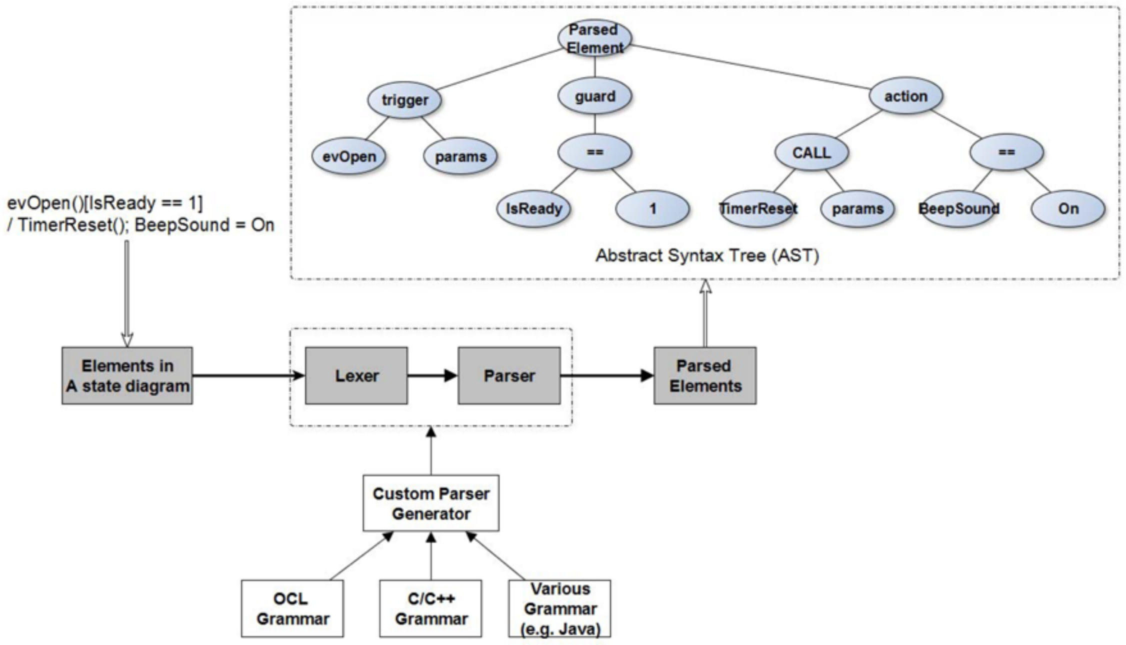


Figure 2. Specification analysis of UML state diagrams using a custom parser generator.

From this paper, suppose that a test data generator can generate test cases from both a platform independent model and a platform dependent model for the usability of the test data generator. If a syntax definition of the natural language is changed or an action language is used to implement the platform independent model, parser should be generated again for this development environment. Therefore, a test data generator is proposed to appropriate for the various changes of the test model using the parser generator.

Unit test cases are generated using the information on the states and transitions available in the state diagrams. First, the triggers and guards are used as the input conditions to generate the necessary input values within the variable range. For example, when the transition has a trigger of "evOpen" and the guard is a Boolean variable such as "isReady," test cases from the input values of false or true can automatically be generated. There can be an enormous number of such input conditions depending on the transition conditions of the state diagrams. In particular, the wide range of input values available could lead to the

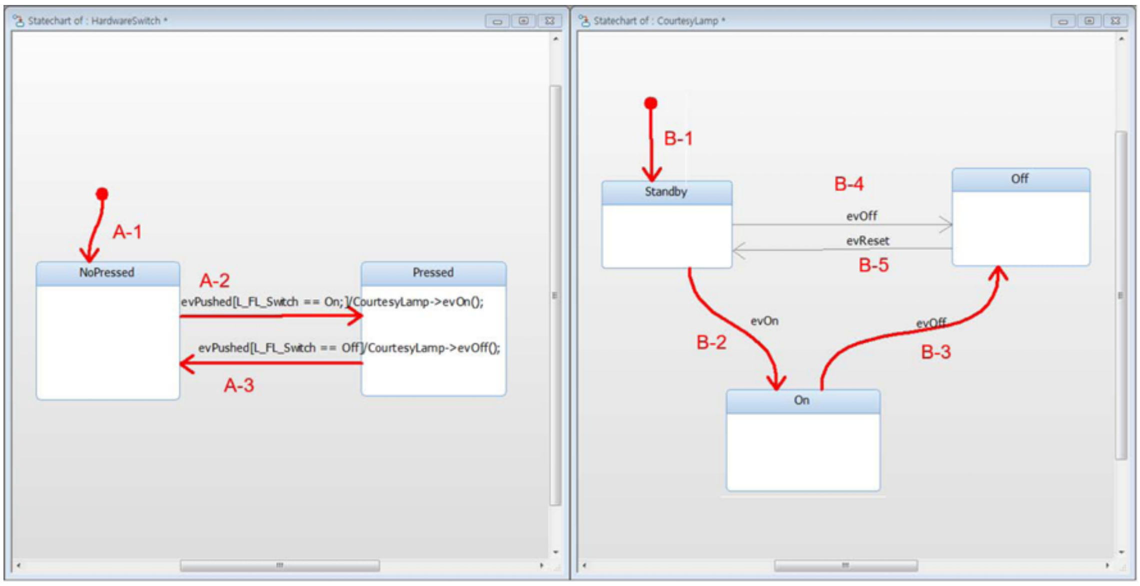

Statechart of : HardwareSwitch *
A-1
NoPressed
A-2
Pressed
evPushed[L_FL_Switch == On;]/CourtesyLamp->evOn();
evPushed[L_FL_Switch == Off]/CourtesyLamp->evOff();
A-3
Statechart of : CourtesyLamp *
B-1
Standby
B-4
Off
evOff
evReset
B-5
evOn
B-2
evOff
B-3
On

Figure 3. Simple example of courtesy lamp statechart.

Table 1. Example of generated unit test cases from each classes.

Unit test cases	Path
A-1	Root(A) → NoPressed
A-2	Root(A) → NoPressed → Pressed
A-3	Root(A) → NoPressed → Pressed → NoPressed
B-1	Root(B) → Standby
B-2	Root(B) → Standby → On
B-3	Root(B) → Standby → On → Off
B-4	Root(B) → Standby →On→Off → Standby
B-5	Root(B) → Standby → On → Off → Standby → Off

generation of a large number of cases and thus take a very long time to process.

Output signals that correspond to a combination of inputs can be analyzed through transition actions. The input and output data corresponding to the paths for test case generation are stored in a record table. This is used to generate expected values and to check for the possibility of transition.

Unit test cases generated from state diagrams may be created using various path search algorithms. We used a breadth-first search algorithm for transition coverage. This traverse process can be continued until all of the test cases are generated covering all of the possible paths, including control statements such as loop and conditional statements, until no better solution is found in the given time.

Figure 3 can be shown as an example how test cases are generated on statechart. As a result, Table 1 is showing transition paths of generated test cases for satisfying the transition coverage.

3.2. Integration Test Case Generation

The automatic generation of integration test cases has been one of the most challenging problems in test automation. It is especially difficult to discern all of the possible paths produced by the complex integrated modules within a given period of time. To solve this problem, many studies have produced results based on genetic algorithms. In general, these reduce the time required to solve nondeterministic polynomial-time complete problems such as the travelling salesman problem. Nevertheless, as the software becomes more complex, the creation of integration test cases also becomes more difficult, to the point where it may not be possible to create a suitable integration test case within a given amount of time.

Unit test cases may include basic input and output signals that can allow each function or module to be tested (Leitner *et al*., 2007). We propose a new method that allows unit test cases to be integrated using the relationship information from the UML. The software is combined from each unit module, and then the integration test cases can be integrated from these. Input and output signals used only for each module must be removed at the integration stage so that internal and external variables in the unit module can be distinguished from each other. In this paper, variables are classified into internal variables and external variables using a specific variable-naming rule and the UML specifications. This method can be considered gray-box testing, which is a combination of black-box and white-box testing.

In general, integration testing approaches can be categorized as either top-down or bottom-up. Although the advantages and disadvantages of each can vary, it is necessary to select the proper integration approach depending on the complexity of the inputs and outputs. For example, if one input can affect many functions, a top-down approach makes it possible to test several functions at any time. If a bottom-up approach is used for the same

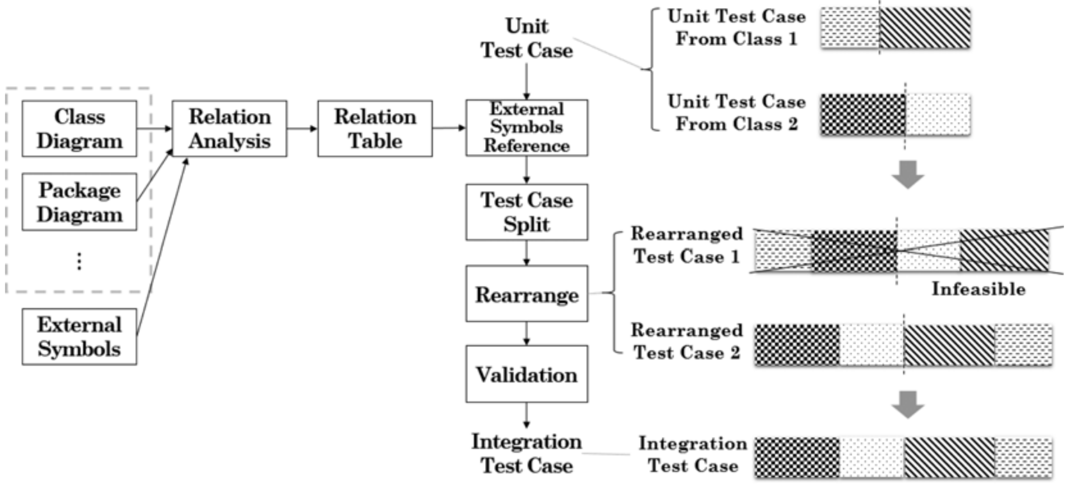


Figure 4. Generation process of integration test cases.

system, it is possible to undertake tests that focus on one particular function, because the focus is on the input conditions relating to the output for a particular function (Myers *et al.*, 2011). As mentioned in the case study of the Power Window Switch Module described in Section 4, a bottom-up integration approach is used for the embedded system, which requires input combinations for the production of many outputs.

First, a base module needs to be selected. Because the bottom-up approach tracks from the base of the modules in the lowest level to the top within the subsystem, a test is needed to determine which base module is to be used.

In general, a class is composed of methods, events, properties, and so on. Relational diagrams such as class diagrams and object diagrams can be used to identify the relationship between each of these classes. Several types of relationship are given in UML, but class diagrams, association, aggregation, composition, and dependency are useful for understanding the relationship between various classes.

Once the relationships between each class have been analyzed, a relation table is created based on the results. The relational information within this table can also be represented as a flow diagram, such as that shown in Section 4 (Figure 7).

In this paper, a divide-and-conquer approach is proposed to generate integration test cases as shown in Figure 4. Test input data for integration testing is divided into individual units of test input and expected values with reference to the flow control. Then, each unit test input and expected value for integration testing is recycled from test input data and the test oracle for unit testing. For this process, unit test cases are divided into partial test cases at the external signal point, and then lower partial test cases begin to integrate upper partial test cases sequentially through the relation table. The test cases that are generated then pass through a process of validation.

From the above example, to enter the path of B-2 shown in the right column of Figure 3, the “evOn” event needs to be generated. If the current path in state diagram A is moved from A-1 to A-2, as shown in left column, the “evOn” event is generated and then the current path in state diagram B can be moved from B-1 to B-2. Therefore, the guard condition for the transition from A-1 to A-2 is given by L_FL_Switch == On. As a result of this combination, generated integration test cases are shown in Table 2.

The UML state machine includes characteristics of both the Moore and Mealy models; however, we defined our state machine based solely on the Mealy model. The output functions can be represented by $\omega = S \times \Sigma \rightarrow \Gamma$, where S is a set of states, Σ is an input character, Γ is an output character, and ω is the output function. The execution path of the state machine can be represented as a sequence of states,

$$P = [s_1, s_2, s_3, \cdots, s_n]$$

where s_1 is the initial state, s_n is the final state, and the range of the passed states is s_i $(1 < i < n)$ (Zeng *et al.*, 2003).

In addition, there are two related state diagrams that have n number of path elements of the upper module P^u, and m number of path elements of the lower module P^l. This can be represented as

$$P^u = [s_1^u, s_2^u, s_3^u, \cdots, s_n^u]$$

$$P^l = [s_1^l, s_2^l, s_3^l, \cdots, s_m^l]$$

Table 2. Example of a newly generated integration test case list.

Generated integration test cases	Path	Validity
A-1 & B-1	ROOT(A) → NoPressed → ROOT(B) → Standby	Valid
A-1 & B-2	ROOT(A) → NoPressed → ROOT(B) → Standby → On	Invalid
A-2 & B-2	ROOT(A) → NoPressed → Pressed → ROOT(B) → Standby → On	Invalid
A-2 & B-2: 2	ROOT(A) → NoPressed → ROOT(B) → Standby → Pressed → On	Valid
A-2 & B-3	ROOT(A) → NoPressed → Pressed → ROOT(B) → Standby → On → Off	Invalid
A-2 & B-3: 2	ROOT(A) → NoPressed → ROOT(B) → Standby → Pressed → On → Off	Invalid
A-2 & B-3: 3	ROOT(A) → NoPressed → ROOT(B) → Standby → On → Pressed → Off	Invalid
A-3 & B-3	ROOT(A) → NoPressed → ROOT(B) → Standby → Pressed → On → NoPressed → Off	Valid
A-3 & B-3: 2	ROOT(A) → NoPressed → ROOT(B) → Standby → Pressed → On → Off → NoPressed	Invalid
A-3 & B-3: 3	ROOT(A) → NoPressed → ROOT(B) → Standby → Pressed → NoPressed → On → Off	Invalid
A-3 & B-3: 4	ROOT(A) → NoPressed → ROOT(B) → Standby → On → Off → Pressed → NoPressed	Invalid
A-3 & B-3: 5	ROOT(A) → NoPressed → ROOT(B) → Standby → On → Pressed → Off → NoPressed	Invalid
A-3 & B-3: 6	ROOT(A) → NoPressed → ROOT(B) → Standby → On → Pressed → NoPressed → Off	Invalid

If the output of the ith ($i < n$) state element of the upper module is related to the input of the jth ($j < m$) state element of the lower module,

$$\omega_i^u : s_i^u \times \Sigma_i^u \rightarrow \Gamma_i^u$$

$$\omega_j^l : s_j^l \times \Sigma_j^l \rightarrow \Gamma_j^l$$

$$R = \Gamma_i^u \cap \Sigma_j^l \neq \varphi$$

then an execution path for the upper module can be divided as follows:

$$P^u = \left[P_{\text{before}}^u, s_i^u, P_{\text{after}}^u\right]$$

$$P_{\text{before}}^u = \left[s_1^u, s_2^u, s_3^u, \cdots, s_{i-1}^u\right]$$

$$P_{\text{after}}^u = \left[s_{i+1}^u, s_{i+2}^u, s_{i+3}^u, \cdots, s_n^u\right]$$

Similarly, an execution path of the lower module is as follows:

$$P^l = \left[P_{\text{before}}^l, s_j^l, P_{\text{after}}^l\right]$$

$$P_{\text{before}}^l = \left[s_1^l, s_2^l, s_3^l \cdots, s_{j-1}^l\right]$$

$$P_{\text{after}}^l = \left[s_{j+1}^l, s_{j+2}^l, s_{j+3}^l \cdots, s_m^l\right]$$

Each of these paths can be integrated into one path, as follows:

$$P^{\text{int egration}} = [P_{\text{before}}^l \cap P_{\text{before}}^u, s_i^u, s_j^l, P_{\text{after}}^l \cup P_{\text{after}}^u]$$

This is represented in Figure 5.

It is considerably difficult to obtain a desired integration test case by recombining unit test cases alone because integration test case generation depends on the generated unit test cases. If the unit test cases for a state machine are generated, these paths can be used to generate integration test cases for all combinational paths.

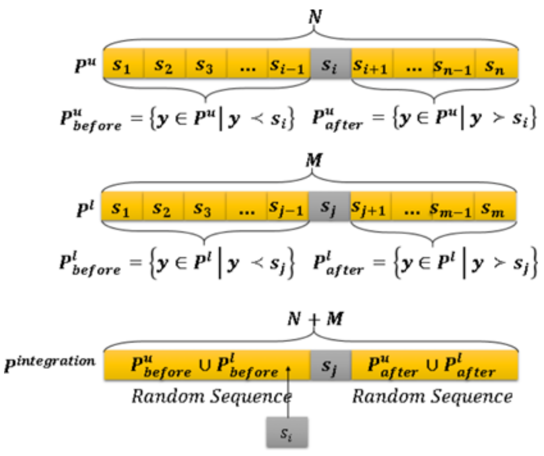

N
P^u
s_1 s_2 s_3 ... s_{i-1} s_i s_{i+1} ... s_{n-1} s_n
$P_{before}^u = \{y \in P^u | y \prec s_i\}$
$P_{after}^u = \{y \in P^u | y \succ s_i\}$
M
P^l
s_1 s_2 s_3 ... s_{j-1} s_j s_{j+1} ... s_{m-1} s_m
$P_{before}^l = \{y \in P^l | y \prec s_j\}$
$P_{after}^l = \{y \in P^l | y \succ s_j\}$
N + M
$P^{integration}$
$P_{before}^u \cup P_{before}^l$
s_j
$P_{after}^u \cup P_{after}^l$
Random Sequence
Random Sequence
s_i

Figure 5. Path-based integration test case generation.

Algorithm 1. Test case integration.

Input: *a set of upper unit test cases P^u, a set of lower unit test cases P^l and relational information Σ_r.*
Output: *a set of integration test cases IPs.*

```
$IPs \leftarrow \Phi$
for each $L \in P^l$ do
  for each $U \in P^u$ do
    $RPs \leftarrow \Phi$
    if IsNotRelated($U, L, \Sigma_r$) then next for
      $SP^u$ = DivideUnitTestCase($U, \Sigma_r$)
      $SP^l$ = DivideUnitTestCase($L, \Sigma_r$)
      $RPs \leftarrow RPs \cup$ Combine($SP^u, SP^l, \Sigma_r$)
      $RPs \leftarrow RPs \cap$ GetValidateSet($RPs$)
      for each $RP \in RPs$ then
        $T \leftarrow$ Initialize temperature
        if IsNotDuplicated($IPs, RP$) then
          $IPs \leftarrow IPs \cup RP$
        end if
        while time limit is not exceeded do
          while $T > 0$ do
            $RP' \leftarrow$ FindNeighbor($RP, T$)
            if IsCoveredNewPath($IPs, RP'$) or
            (Probability($RP', RP, T$) > random(0,1))
            then
              $RP \leftarrow RP'$
            end if
            if IsNotDuplicated($IPs, RP$) and
            IsValidate($RP$) then
              $IPs \leftarrow IPs \cup RP$
            end if
            end while
            $T \leftarrow$ Decrease($T$)
          end while
  end while
  end for
end for
return $IPs$;
```

However, when there are many unit test cases, it is inefficient to use all of them to generate integration test cases. To solve this problem, our method deletes infeasibly combined test cases and explores the neighbors of the generated integration test cases using a simulated annealing algorithm (Bertsimas and Tsitsiklis, 1993). This process is summarized in Algorithm 1.

This generation method can reduce the time needed for generating integration test cases because of the recombination of paths and input/output signals from the unit test cases, compared to typical random input generation methods. However, the main disadvantage of

Table 3. Example of a hardware mapping table.

Name	Type	Hardware	Location	Direction	Min	Max
LogicIgnition	Boolean	Digital I/O card	PortA.0	Input	0	1
WindowDownSwitch	Boolean	Digital I/O card	PortB.0	Output	0	1
AnalogVoltage	Integer	Analog output card	Channel 1	Input	0	1023
DriverDoorClosed	Integer	CAN interface	600h.12.2	Input	0	255
PwmLampBrightness	Double	Counter/Timer card	Channel 1	Output	0	100

this method is that it can lead to an insufficient number of integration test cases when there is not a variety of unit test case paths and input signals. The low probability of random mutations can lead to new test input combinations that can be used to explore unknown areas.

3.3. Test Case Conversion and HIL Testing

To do the hardware testing, it is necessary to convert the software variables to the appropriate hardware signals. For example, the hardware target can be tested via digital/ analog IO devices and several communication interfaces. In this process, unlike the software variables, hardware signal delays can occur when passed through other internal circuits. In the case of manual test case generation, this delay is important. In this paper, hardware test cases were automatically generated from software integration test cases, and the hardware signal delay was taken into consideration in the conversion process.

As shown in Table 3, hardware mapping tables are used to store the hardware and software mapping information and make it possible to keep changes in hardware test cases to a minimum when the hardware test environment is altered. In other words, such tables are used to convert software integration test cases into hardware test cases. In addition, they contain information such as the type of test instruments that can be connected to the hardware, and the type of input and output signals (e.g., digital, analog, and pulse-width modulation). This information is changed according to the target and test instruments used when the test environment is changed. If the change in hardware environment occurs frequently without the need to convert the software test cases in advance, it can be more efficient to convert into a hardware test case at runtime, because it is better to modify the table rather than all of the hardware test cases whenever the environment changes.

Unlike software tests, hardware tests include a delay between the signals. For example, in the case of TTL-compatible digital input signals, “0” and “1” can be seen to change on the rising edge from zero to five Volts through a reasonable time delay. For this reason, it is necessary to insert a time delay between the continuous inputs of a signal so that the test target can be properly processed. In addition, if necessary, an appropriate time delay needs to be inserted between signals not only for the continuous input of one signal, but the continuous inputs of multiple signals. Moreover, even if time is needed to get the output value corresponding to the input combination, a time delay may be inserted between the signals through a conversion process like Table 4.

Table 4. Conversion process of software variables into hardware signals.

Case	Before changing signals	After changing signals

Change of value for a continuous input signal	IgnitionOn.Write(0);
	IgnitionOn.Write(1);
		IgnitionOn.Write(0);
		Wait(100);
		IgnitionOn.Write(1);

Continuous input from several signals	IgnitizonOn.Write(1);
	AccessoryOn.Write(1);
	InputSwitch.Write(1);
		IgnitionOn.Write(1);
		Wait(100);
		AccessoryOn.Write(1);
		Wait(100);
		InputSwitch.Write(1);

Checking output after specific inputs	IgnitionOn.Write(1);
	OutputSignal.Read(1);
		IgnitionOn.Write(1);
		Wait(100);
		OutputSignal.Read(1);

The hardware mapping table is saved in XML format and can be used for the conversion. To facilitate the conversion process, extensible stylesheet language transformations (XSLT) and XQuery can be used (Shin *et al.*, 2013). XSLT is seen as a standard for the transformation of XML documents and can transform XML documents into other types of document (Kay, 2007). On the other hand, XQuery is a query language for XML documents that can flexibly retrieve the necessary information from XML documents (W3C, 2010). The advantage of each method for conversion may vary depending on the case, but we used XSLT because it more easily retrieves information from small files.

4. CASE STUDY: POWER WINDOW SWITCH MODULE

We applied our system to the power window switch module of a Hyundai Santa Fe vehicle. First, a UML model was implemented using IBM Rational Rhapsody. Power window switch behavior was implemented using a state diagram. While UML models are usually made as

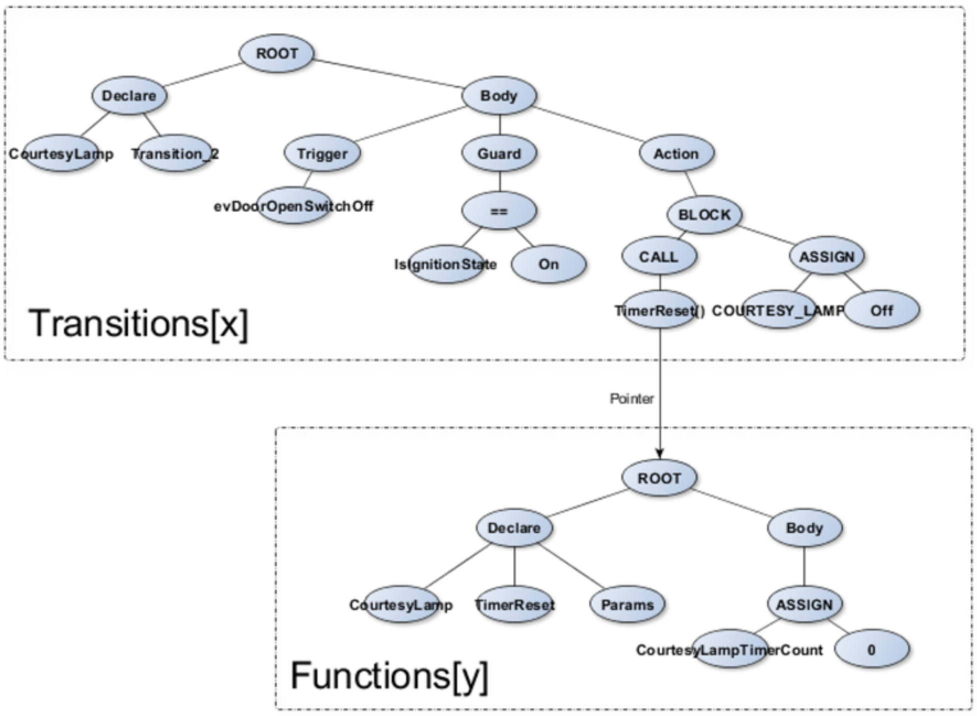


Figure 6. Example of an abstract syntax tree.

platform-independent models (PIM), some UML tools for automatic code generation are able to implement platform-specific models using a source-level language (Douglass, 2002). In this case, we used C. Then the UML model was converted into XMI. Because the platform-specific model uses the state diagram specification in C and is mixed with the PIM into XMI, an software structure of test case generator is proposed so that the various specified languages can be parsed by the parser generator, including support for language-dependent custom grammar rules.

C language syntax is parsed by a custom parser, and is represented by the abstract syntax tree shown in Figure 6.

This tree is used to generate test cases. In general, source-level languages in C include control statements such as condition statements and loop statements. These statements comprise relational operators, conditional operators, numerical operators, variables, constants, and so on. After the parser has analyzed these elements, test cases are generated based on branch coverage. In particular, to generate a valid test case, the test input values must be changed within a meaningful range. To reduce the search space, in the case of a variable that uses the enumerator, the test case input values are generated from the enumerator list. If a suitable path cannot be found, the input value space is extended to cover the entire range of the variable type.

Software test cases generated via a model are used to measure the code coverage. If there are any errors present in the source code (because the test case is generated from the source code) the test case can be made to show these errors. However, the advantage of our method is that it can detect an error that occurs during coding using a generated test case from a model.

Code coverage analysis was performed using VectorCAST (one of a number of commercial tools used for dynamic testing). Software test cases in XML format were converted into a comma-separated values (CSVs) files format, and then imported into VectorCAST. Code coverage was analyzed during unit testing and produced a test result that can be compared to the expected value (Shin *et al*., 2014).

The power window switch module includes a number of functions in addition to the window control, depending on the model. For example, the module used in the Santa Fe DM controls the side mirror, courtesy lamp, puddle lamp, and heater. We applied our method to produce a thousand test cases to achieve branch coverage in the power window switch module of the Santa Fe DM. Of course, the number of test cases might be different from this depending on the target code coverage and optional add-on features.

The unit test cases generated can be utilized to generate other integration test cases. To integrate cases, each unit module has to be integrated. First, the relationship between each unit module needs to be identified using a relational diagram such as a class diagram or an object diagram. If a tester selects the lowest node as a reference module, the relationship between each module can be analyzed using a bottom-up approach and starting from the reference module.

After conducting relationship analyses, integration test

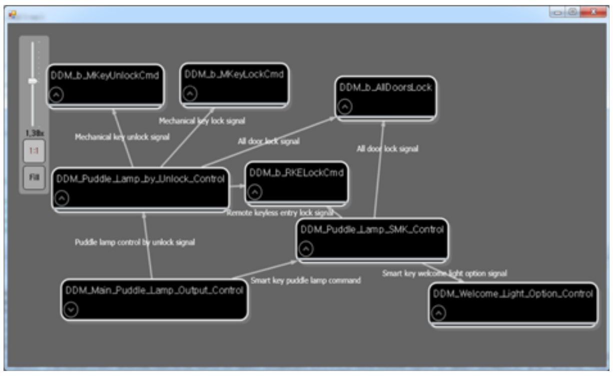

1.38x
1:1
Fill
DDM_b_MKeyUnlockCmd
DDM_b_MKeyLockCmd
DDM_b_AllDoorsLock
Mechanical key lock signal
Mechanical key unlock signal
All door lock signal
All door lock signal
DDM_Puddle_Lamp_by_Unlock_Control
DDM_b_RKELockCmd
Remote keyless entry lock signal
DDM_Puddle_Lamp_SMK_Control
Puddle lamp control by unlock signal
Smart key welcome light option signal
Smart key puddle lamp command
DDM_Main_Puddle_Lamp_Output_Control
DDM_Welcome_Light_Option_Control

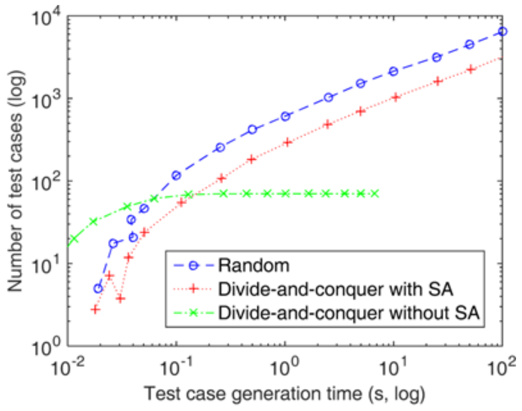
Figure 7. Module flow diagram of a puddle lamp.

cases were created on the basis of the information gained from these analyses. Because each module was considered a flow-based association, the program created a relationship diagram in terms of a module flow diagram as shown in Figure 7. In this case it is better to decide the variable names using a specific naming rule because it is difficult to distinguish between internal and external variables. Other variables can be considered actual input and output signals of the integrated module and can act as a stimulus for the integrated test cases.

As mentioned in Section 3.2, after the unit test cases are divided during the integration process based on the point at the external symbol, all related unit test cases are integrated step-by-step, starting from the reference node. To eliminate infeasible test case after integration, the program references a test oracle, which is generated for unit testing based on model. Finally, the remaining test cases are recycled again to integrate the other upper modules.

We compared the results using our system to those using automatic generated path-based test cases. The generation algorithm was developed using C# and the recorded times were obtained using a PC with an Intel Xeon E3-1231 v3 with a 3.4 GHz processor.

Each experiment was repeated 10 times. The upper side of Figure 8 shows how the number of test cases is related to the running time of each algorithm. As a system under test (SUT), the courtesy lamp is related with two classes, and two state diagrams that contain eight states and a dozen transitions. It has a function of eight input parameters, including a logic input value and time variable, increasing every 0.1 seconds (time step). First, the divide-and-conquer generation algorithm without simulated annealing could not explore other test cases because integration test cases are generated only from unit test cases, and it has insufficient code coverage – as shown at the below side of Figure 8. On the other hand, random data generation and a divide-and-conquer generation algorithm with simulated annealing generate a number of test cases that increase proportionally to the execution time. The divide-and-conquer method does not produce a sufficient number of test cases during a short execution time due to the overhead



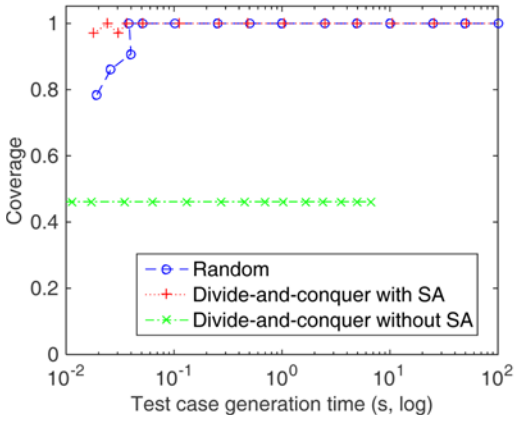


Figure 8. Performance comparison of integration test case generator for courtesy lamp.

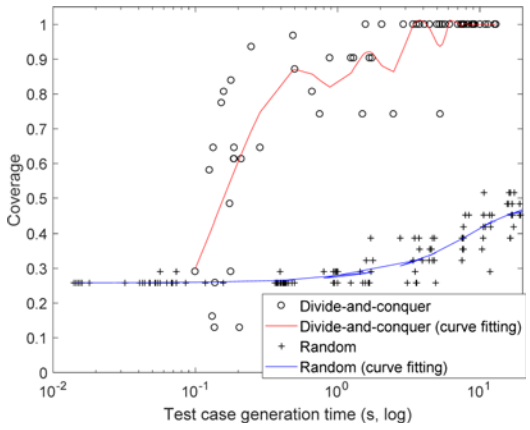


Figure 9. Performance comparison of integration test case generator for puddle lamp.

corresponding with the division of unit test cases before the generation of integration test cases. Figure 9 shows the result of integration test case generation on the courtesy lamp, which has eight classes and state diagrams with over a hundred variables. The range of integer values was set from 0 to 65535. In the random input generation method, code coverage is below 50 % even after 10 minutes.

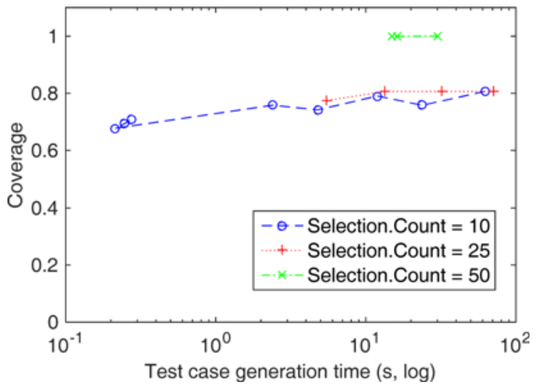


Figure 10. Parameter effect on the unit test case selection using divide-and-conquer.

Table 5. PXI-based testing instruments used to test the power window switch module.

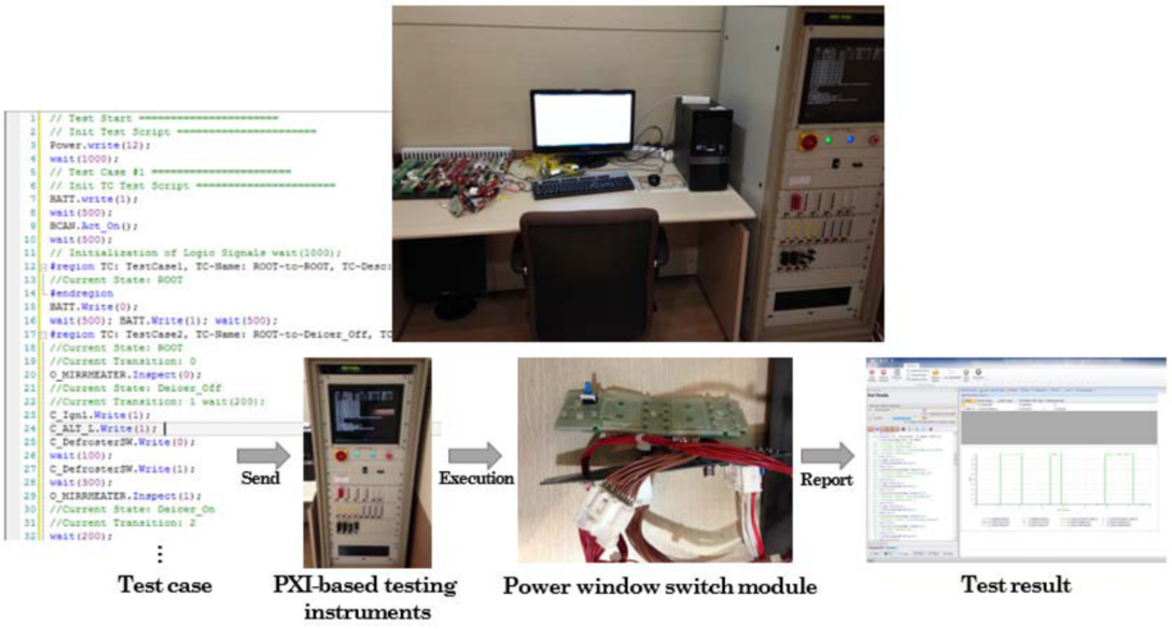
Device name	Description
PXI-8110	2.26 GHz quad-core PXI-embedded controller
PXI-1044	14-slot 3U PXI chassis
PXI-8432/4	Serial interface
PXI-8513/2 × 2	CAN interface
PXI-6143 × 2	16-bit multifunction DAQ
PXI-6733	16-bit, 8 channel, analog output
PXI-6533 × 2	32 digital input and output
PXI-6624	8 channel counter/timer

However, divide-and-conquer approaches with simulated annealing can generate a lot of integration test cases, and their code coverage is 100 %. Generally, the performance of proposed approaches was better than the random input generation on the power window switch module by at least 30 %.

The number of generated test cases and their execution time changed according to the number of unit test cases, as show in Figure 10. If unit test cases are selected over 100, transition coverage is 100 % because various combinations of integration test cases can be generated – but it is too slow. In that case, the unit test cases are selected only below 10; although the execution time is short, it is not enough to combine new integration test cases for code coverage. The choice between these parameters should be made depending on the size of the problem.

Then we used our method to generate hardware test cases. To this end, irrelevant variables must be eliminated from the test cases and then the cases must be converted into hardware test cases composed of actual hardware signals, such as digital and analog signals. These are mainly eliminated during the integration process. At the end of this process, the final remaining variables are converted into input and output signals needed for hardware testing.

Note that a suitable time delay must be considered for the hardware application. As described above, these are added between signals after the information related to the time delay is stored in the mapping table. In addition, other commands such as device initialization may be required before and after testing. A hardware mapping table is



```
// Test Start ======================
// Init Test Script ======================
Power.write(12);
wait(1000);
// Test Case #1 ======================
// Init TC Test Script ======================
BATT.write(1);
wait(500);
BCAN.Act_On();
wait(500);
// Initialization of Logic Signals wait(1000);
#region TC: TestCase1, TC-Name: ROOT-to-ROOT, TC-Desc:
//Current State: ROOT
#endregion
BATT.Write(0);
wait(500); BATT.Write(1); wait(500);
#region TC: TestCase2, TC-Name: ROOT-to-Deicer_Off, TC
//Current State: ROOT
//Current Transition: 0
O_MIRRHEATER.Inspect(0);
//Current State: Deicer_Off
//Current Transition: 1 wait(200);
C_Ign1.Write(1);
C_ALT_L.Write(1);
C_DefrosterSW.Write(0);
wait(100);
C_DefrosterSW.Write(1);
wait(300);
O_MIRRHEATER.Inspect(1);
//Current State: Deicer_On
//Current Transition: 2
wait(200);
⋮
```

Figure 11. Hardware testing process for the power window switch module.

created so such commands can be automatically added during the conversion process.

To verify which power window switch modules used the generated hardware test cases, we used National Instruments real-time PXI-based instruments as outlined in Table 5. The converted test cases were transferred to the test instruments via the network and then executed for hardware testing. This entire hardware test process is represented in Figure 11.

If the test instruments receive a hardware test case from the host computer, the instruments perform a hardware test according to the test case input and output stimuli, and send the results to the host computer. This process differs from a general HIL test system because the test cases and their results are transmitted across the network. The names of the input and output stimuli in the files are stored as software variables. Each variable name is then converted via a hardware mapping table into a signal for use by the hardware as soon as the test case is transmitted across the network. Then these test cases can be managed independently of the hardware, and it is possible to construct a more flexible test environment. Although test evaluation environments that use networks can be self-constructing, we used a commercial program (Test Executor of BTS Technologies Inc.) for execution and evaluation.

5. CONCLUSION

Our proposed process generates software and hardware test cases for embedded systems using a UML model. The UML model is first converted into XMI format to generate test cases, and then state diagram specifications are converted into an abstract syntax tree using a custom parser generator. The unit test cases taken from the AST and state diagrams are generated based on breadth-first searching. This proposed method generates proper test cases without any dependence on a specific source-level language (C/C++, java).

Next, we integrate the unit test cases, using a new divide-and-conquer approach based on the bottom-up relationships of each module. Because this approach reuses the path and input/output information from each unit test case it reduces the time required to generate an integration test case compared to random input generation.

Finally, integration test cases are converted back into hardware test cases. For this process, software test cases need to be converted according to the particular hardware test environment. We used XSLT for this. Then, utilizing the transformed hardware test cases, the software in the embedded system is tested by means of PXI-based hardware test instruments. We applied our approach to the power window switch module of a Hyundai Santa Fe vehicle.

Our method successfully automates the overall software and hardware tests for embedded software, helping to minimize the resources needed during the test phase. It also allows for modifications without increasing the amount of processing time because the integration test cases are generated from the unit test cases. This could be improved dynamic testing and make it possible to further automate the generation of hardware test cases. However, our method cannot be used for all types of test; for example, it is not possible to create a test case based on human experience, e.g., fault injection testing. Nonetheless, our method should prove useful in preventing human error during the manual generation of test cases.

ACKNOWLEDGEMENT–This work is supported by the GRRC program of Gyeonggi province (GRRC Hanyang 2014-B03) and also by World Class 300 project (10050405) of the Ministry of Trade, Industry and Energy (MOTIE) and the Small and Medium Business Administration (SMBA) of South Korea.

REFERENCES

Anand, S., Burke, E., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., Mcminn, P. and Bertolino, A. (2013). An orchestrated survey on automated software test case generation. *J. Systems and Software* **86**, **8**, 1978−2001.

Beizer, B. (2003). *Software Testing Techniques*. Dreamtech Press. New Delhi, India.

Bertolino, A. (2001). *Guide to the Software Engineering Body of Knowledge − SWEBOK*. IEEE Press. Washington, USA.

Bertsimas, D. and Tsitsiklis, J. (1993). Simulated annealing. *Statistical Science*, **8**, 10−15.

Bringmann, E. and Kramer, A. (2008). Model-based testing of automotive systems. *IEEE 1st Int. Conf. Software Testing, Verification, and Validation*, 485−493.

Chen, T. Y., Poon, P.-L., Tang, S.-F. and Tse, T. (2012). DESSERT: A divide-and-conquer methodology for identifying categories, choices, and choice relations for test case generation. *IEEE Trans. Software Engineering* **38**, **4**, 794−809.

Douglass, B. P. (2002). Model Driven Architecture and Rhapsody. Technical Report. I-Logix.

Florin Pinte, F. S. and Norbert, O. (2008). Automatic generation of optimized integration test data by genetic algorithms. *Software Engineering Workshops (W. Maalej, B. Bruegge (Hrsg.))*, Gesellschaft für Informatik, Bonn, Germany.

Fraser, G. and Wotawa, F. (2007). Using LTL rewriting to improve the performance of model-checker based test-case generation. *Proc. 3rd Int. Workshop Advances in Model-based Testing*, London, UK, 64−74.

Gulia, P. and Chillar, R. S. (2012). A new approach to generate and optimize test cases for UML state diagram using genetic algorithm. *ACM SIGSOFT Software Engineering Notes* **37**, **3**, 1−5.

Hartmann, J., Imoberdorf, C. and Meisinger, M. (2000).

UML-based integration testing. *ACM SIGSOFT Software Engineering Notes* **25**, **5**, 60–70.

Heumann, J. (2001). Generating Test Cases from Use Cases. http://students.mimuw.edu.pl/~zbyszek/posi/GeneratingTestCasesFromUseCasesJune01.pdf

Kay, M. (2007). Xsl Transformations (xslt) Version 2.0. http://www.w3.org/TR/xslt20

Shin, K.-W., Kim, S. S. and Lim, D.-J. (2013). Automatic test-case generation for hardware-in-the-loop testing of automotive body control modules. *SAE Paper No*. 2013-01-0161.

Lefticaru, R. and Ipate, F. (2007). Automatic state-based test generation using genetic algorithms. *IEEE Int. Symp. Symbolic and Numeric Algorithms for Scientific Computing, SYNASC*, 188–195.

Leitner, A., Oriol, M., Zeller, A., Ciupa, I. and Meyer, B. (2007). Efficient unit test case minimization. *Proc. Twenty-second IEEE/ACM Int. Conf. Automated Software Engineering*, Atlanta, Georgia, USA, 417–420.

Myers, G. J., Sandler, C. and Badgett, T. (2011). *The Art of Software Testing*. John Wiley & Sons. Hoboken, New Jersey, USA.

Offutt, J. and Abdurazik, A. (1999). *Generating Tests from UML Specifications*. «UML»'99 – The Unified Modeling Language. Spriger-Verlag Berlin Heidelberg. Heidelberg, Germany.

Ogata, S. and Matsuura, S. (2010). A method of automatic integration test case generation from UML-based scenario. *WSEAS Trans. Information Science and Applications* **7**, **4**, 598–607.

Samuel, P., Mall, R. and Bothra, A. K. (2008). Automatic test case generation using unified modeling language (UML) state diagrams. *IET Software* **2**, **2**, 79–93.

Samuel, P., Mall, R. and Kanth, P. (2007). Automatic test case generation from UML communication diagrams. *Information and Software Technology* **49**, **2**, 158–171.

Shafique, M. and Labiche, Y. (2010). A Systematic Review of Model Based Testing Tool Support. Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04.

Shin, K., Kim, S., Park, S. and Lim, D. (2014). Automated test case generation for automotive embedded software testing using XMI-based UML model transformations. *SAE Paper No*. 2014-01-0315.

W3C (2010). XQuery 1.0: An XML Query Language. Second Edition ed.

Windisch, A., Wappler, S. and Wegener, J. (2007). Applying particle swarm optimization to software testing. *Proc. 9th Annual Conf. Genetic and Evolutionary Computation*, London, UK, 1121–1128.

Hartig, W., Habermann, A. and Mottok, J. (2009). Model-based Testing for Better Quality. http://www.vector.com/portal/medien/cmc/press/PND/Modellbasiertes_Testen_ElektronikAutomotive_200903_PressArticle_EN.pdf

Tung, Y.-W. and Aldiwan, W. S. (2000). Automating test case generation for the new generation mission software system. *Proc. IEEE Aerospace Conf.*, 431–437.

Zelkowitz, M. V. (1978). Perspectives in software engineering. *ACM Computing Surveys (CSUR)* **10**, **2**, 197–216.

Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J. and Sheng, Q. Z. (2003). Quality driven web services composition. *Proc. 12th Int. Conf. World Wide Web*, Budapest, Hungary, 411–421.

Zhan, Y. and Clark, J. A. (2005). Search-based mutation testing for Simulink models. *Proc. 7th Annual Conf. Genetic and Evolutionary Computation*, Washington DC, USA, 1061–1068.