Automatic Test Cases Generation based on State chart

Jiawei Wang (jw53477) jiawei.wang@utexas.edu

Jun Wang (jw53468) wangjun1127@utexas.edu

Yiqian Zhang (yz6834) irenezhang@utexas.edu

ABSTRACT

Test cases design and implementation have always been a time-consuming task for the software development process. To generate comprehensive test cases automatically is in high demand these days. Statechart diagram, as a critical component in software design processes, is usually neglected during testing. In this paper, we proposed a method to generate distinctive test cases based on a statechart. Our approach combines statechart with a constraint input file provided by users to generate custom defined test case suite automatically. The algorithm contains four stages: statechart parsing, transition sequence generation, invalid path filtering, and test case generation. The algorithm also analyzes path coverage and boundary test for business logic.

Keywords: State Chart Diagram, Test Cases, Junit test, Coverage test, Automation

1. INTRODUCTION

During the process of software development, designing comprehensive coverage test cases has always been a foremost issue for software testing engineers. Testing is taking more and more time compared to other processes. Unit testing is the basis among other different kinds of tests. However, to design a comprehensive unit test suit takes a lot of time and effort, especially when corner cases and coverage analysis are taken into consideration.

Besides, TDD(Test Driven Development) has gained popularity as a methodology for software development. TDD gives organizations the ability to painlessly update their software to address new business requirements or other unforeseen variables. However, this requires developers to design fully documented and inclusive tests before implementing the basic functionality of a system.

As a leading software developing component for software design, a statechart for a service indicates all the possibility of a particular business logic. Thus, statechart is indispensable in software development. In this paper, we proposed an algorithm to generate test cases based on a fully documented statechart. Each test case we generated suggests a specific user scenario and its corresponding path condition. By applying our method, we reduce the barrier between software developers and software testing engineers. This also leads to an improvement of efficiency and consistency in the software development process.

.

In the next few sections, Section 2 discusses the design and implementation for our algorithm. After that, Section 3 describes an experiment for our method. Next, evaluation and results for our algorithm were analyzed. Section 4 briefly summarized related work that we found during our research. Finally, Section 5 and 6 analyze potential areas that we can still work on and concludes our methodology.

2. DESIGN AND IMPLEMENTATION

2.1 Design

2.1.1 Statechart to Java Data Structure

The initial step of an automatic test case generation requires acquiring the state and transition information from a UML statechart diagram. Properly saving the information in a Java data structure enables the analysis of the statechart in later stages. To accomplish this goal, the design employs StarUML[] that converts a UML statechart to an XMI file and uses a custom Java program to parse the XMI file into appropriate data structures.

To supplement the statechart, the java program considers a constraint file to restrict the test case generation space. The constraint file specifies predicates, formulas, sequence order, and path selection. Before describing the test case generation algorithm, we first define the following constraints:

Predicate. A predicate is a requirement that must be met to trigger a transition. That is, the result of a valid predicate should be a boolean value. Each predicate is associated with a transition name in the constraint file.

Formula. Each formula defines an equation for calculating a variable when the variable has a dependency on other variables.

Sequence order. In the real world, a sequence of transition may be invalid even if it adheres to the statechart. With the help of sequence order constraints, our program can filter out invalid sequence properly.

Path selection. Path selection defines the range of a variable used in a specific transition so that the transition can be taken. If the variable is out of the given range, the transition will never be chosen by our program.

2.1.2 Java Data Structure to Transition Sequences

After translating the UML statechart, all the data will be saved into a custom defined data structure. The next step is to traverse all the paths in statechart to obtain all possible transition sequences. Transition in statechart works as

a function that transfers from one state to another. A transition sequence is a collection of transitions on one specific path that a state goes through from the start state to the end state. Generating all the sequences can be seen as an exhaustive search of all possible paths from the statechart. Under certain circumstance, there will be circular transitions, which we defined as state transfers to other states and finally traverses back to itself. Setting a cycle count variable helps to control the number of cyclic transitions we encountered. The total sequence size tremendously relies on this parameter.

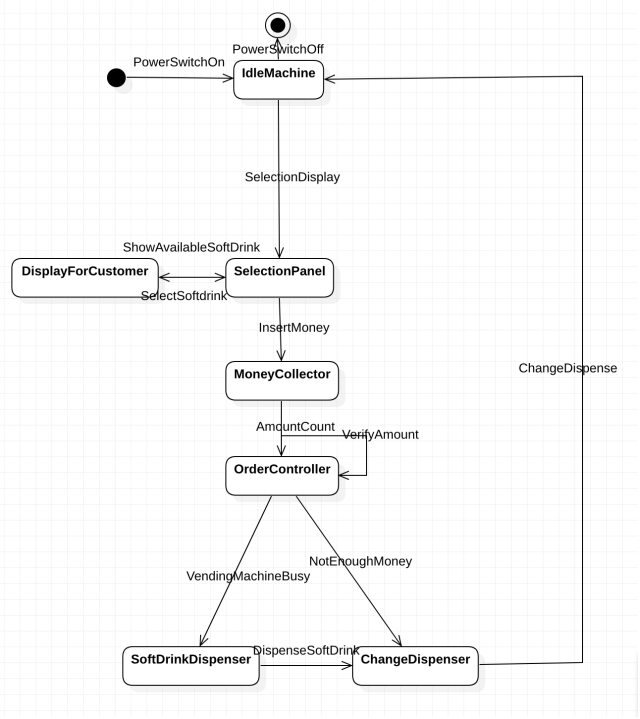

Figure 1: A simple example of a statechart with cycle

As is shown in **Figure 1**, "*PowerSwitchOn -> PowerSwitchOff*" would be the simplest transition sequence in this statechart. Some other complex transition sequences would be like "*PowerSwitchOn -> SelectionDisplay -> ShowAvailableSoftDrink -> SelectSoftdrink -> InsertMoney -> AmountCount -> VerifyAmount -> VendingMachineBusy -> DispenseSoftDrink -> ChangeDispense -> PowerSwitchOff*". There are in total 9 paths from this statechart when cycle count is set to 1. It means a transition sequence traverses 0 or 1 time of the cycle.

A transition sequence list as illustrated above will be generated after processing. One thing to mention is that the validity of the paths we generate will not be taken into consideration at this stage. The filtering process will be discussed in the next section.

2.1.3 Invalid Transition Sequences filtering

All possible transition sequences are generated from the previous step. Nevertheless, some transition sequences may contradict with real business logic or cannot be achieved at the test cases generation step. A filtering method is a promising solution to solve this problem. We define two types of essential invalid sequences at this step. One is an invalid order transition sequence, and the other is a sequence that relies on an indispensable variable.

Specifically, some possible transition sequences could never happen in a real-life business process because the order of those transitions is incorrect. For example, when buying drinks from a vending machine, the user should select drinks before inserting money. However, the insert money action may happen before the action of drink selection in some transition sequences generated from a statechart. These sequences could lead to logic confusion problems when testers are implementing these sequences on a real vending machine. If a user inserts the money first and then select drinks that could have been sold out, the user will not get the drink and the money returned because the vending machine will go to the end state directly after checking the number of available drinks. In order to resolve this problem, the users should suggest a set of constraints of transition orders in the supplemental data input file. Then, the filtering method will eliminate invalid transition sequences according to the constraints provided by the users.

Moreover, generating test cases for the sequences can be difficult because of lacking predicates of indispensable variables. Because the test case generation step uses predicates in transitions to assign an initial value to variables. Particularly, sequences from the state chart of vending machine may involve several variables, such as the total amount of the inserted money and the returned money. The value of the returned money should equal the value of the inserted money minus the total money needed for drinks. However, if a sequence only contains a predicate of the returned money, the test generation step cannot generate a test case for this sequence because the value of the returned money is calculated using the value of the inserted money. Therefore, a sequence lacking predicates of an indispensable variable is considered as an invalid sequence. In order to filter these sequences, the filtering method will identify whether a variable can exist independently. When a sequence contains a dependent variable, and this sequence does not consist of all predicates for the involved variables, an invalid transition flag will be applied to this sequence and our algorithm will eliminate this path in the further steps.

2.1.4 Test Case Generation for Transition Sequences

After filtering all invalid sequences, the next step is to generate test cases automatically. The generation method can be divided into two parts. The first part is to assign an initial value randomly for each variable in a sequence. The second part is to adjust these values until all formulations satisfy the relationship with each other.

When the test case generation method obtains all values of variables, our method sets all values of variables to be the input of a test case and put the current sequence to be the output. Because statecharts describe the business logic of the whole system rather than a specific function, test cases generated from statecharts test the business logic of the entire system. Thus, the output of a test case is a business logic sequence representing a possible business process rather than a specific value or variable.

2.2 Implementation

2.2.1 Parsing Statechart to Java Data Structure

In our implementation, a UML statechart diagram is represented by exactly one instance of the âĂIJStateChartâĂİ class. To enable the traversal of a statechart, the statechart should have a definition of its start state, end state, and a map that stores pairs of the state and the transition list.

An XMI file is taken as an input for our parsing process. This XMI file exactly stores all the information needed for further steps. Meanwhile, a corresponding data input regulation file is provided and served as the constraints for test generation. We define a Statechart data structure which saves all the information regarding the statechart itself and the constraint file. So far, we have successfully parsed the external information to our custom data structure.

2.2.2 Transition Sequences Generation

To generate all transition sequences, our data structure defined in the previous step can be taken as a directed graph. The start state and the end state are the start vertex and the end vertex of the graph. A transition sequence maps to a path from all the edges connection from a start vertex and the end vertex. There exists a lot of algorithms and tools that we can use to traverse all the paths like Java PathFinder(JPF), depth-first search(DFS) and breadth-first search(BFS). To make our work less complicated, we use DFS and backtracking to implement our algorithm. A `StateMachine` class was used to take our custom defined data structure as an input and a transition sequence list as an output.

As mentioned in the design part, a `CycleCount` is used in our implementation to control circular transitions. `CycleCount` accounts for how many times our algorithm keeps going when there is a cycle. This parameter is set to 1 to make the process simple and concise. DFS helps to search through all the possible paths and a corresponding transition sequence is recorded during executing DFS. When the current state reaches the end state, a complete transition sequence will be replicated to the result set. Backtracking is used here to traverse back from the current state to the previous state and exhaustively search for the other states that are not executing yet. **Algorithm 1** is the pseudo-code for generating all transition sequences.

2.2.3 Invalid Transition Sequences filtering

In the design part, we illustrate two different kinds of invalid sequences and the corresponding algorithm to filter them. Thus, two methods are implemented separately.

First, a filter method detects invalid transition order according to an assistant data input file. Specifically, we suggest several certain orders in the assistant data input file to constrain an order in each sequence. A sample of constraint is shown in **Figure 2**. In the **Figure 2**, the order constraints part articulates that the transition of drink selection must happen before the transition of insert money. Therefore, the filter method will search all positions of these predefined transitions. When their positions in a sequence violate the constraint, this sequence will be eliminated from the sequence set. For example, there is a possible sequence that the position of insert money transition existing before the position of drink selection transition. Thus, this sequence will be labeled invalid from the sequence set and not be taken into consideration for further steps.

Second, the filter method will filter sequences lacking essential variables. As shown in **Figure 1**, a statechart of a

Algorithm 1 Generating Transition Sequences

```
1: function Process()
2:     DFSBackTracking(new Sequence, new countMap)
3:     return Results
4: end function
5:
6: function DFSBackTracking(sequence, countMap)
7:     if currentState == EndState then
8:         Add sequence to Results
9:     end if
10:    for transition t from currentState.transtionList do
11:        Add t to sequence
12:        if countMap[t] == CycleCount then
13:            Remove last transition from sequence
14:        end if
15:        // DFS
16:        countMap[t] += 1
17:        currrentState := t.NextState
18:        // Backtracking
19:        countMap[t] -= 1
20:        currentState = t.FromState
21:        Remove last transition from sequence
22:    end for
23: end function
```

```
Order:
SelectSoftdrink InsertMoney
InsertMoney VerifyAmount
ShowToSelectPanel InsertMoney
```

Figure 2: A sample transition order constraints part of a data input file

vending machine, some variables are involved in the business process. For example, when users buy a drink from a vending machine, users will insert money to the machine and then obtain return money and get the drink. In this process, the returned money and the inserted money are two variables involved in this scenario. Actually, the return money is calculated from the inserted money minus total money of drinks. Therefore, variables do not exist independently. In order to obtain relationships of variables, we define a formulation constraint in the assistant data input file for users to suggest relationships among different variables. A sample formulation part is shown in **Figure 3**.

```
Formulation:
ReturnMoney = Amt - TotalMoney
TotalMoney = N * P
```

Figure 3: A sample formulation showing relationships between variables

In **Figure 3**, the variable N refers to the number of drinks bought by users and variable P refers to the price of each drink. In the formulation part, all formula should be written as an equation like A = B + C. Thus, all variables appearing in the left of the equation depends on the value of the

right expression. In other words, all variables appearing in the left expression cannot exist independently because they are calculated from expressions on the right expression. For instance, if we do not know the number of drinks bought by users or prices of drinks, we cannot calculate the total money of drinks. Thus, if a sequence only contains the predicate of dependent variables, but the sequence does not contain all predicates of variables used to calculate the dependent variables. This sequence will be considered as an invalid sequence. Another example would be when a sequence contains a predicate of the total money variable, but it does not consist a predicate of the number of drinks or a predicate of the price. This sequence will also be considered invalid.

2.2.4 Test Case Generation for Transition Sequences

Test case generation method contains two steps. First, the method assigns a random value for each variable involved in a sequence. Second, the method adjusts these values until satisfying formulations in the assistant data input file. In order to explain the method clearly, we propose a sample state chart shown in **Figure 4**. The proposed state chart contains two sequences from state 1 to 5 and to 3. Additionally, a formulation c = b - a showing a relationship between the variable in the text box. The following explanation based on the sequence from state 1 to state 5.

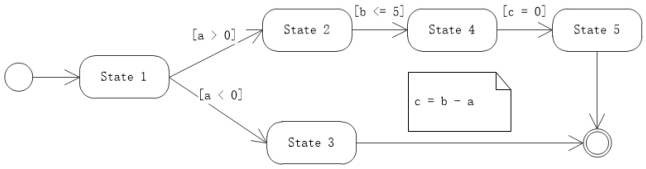

Figure 4: A sample state chart

First, the method assigns a random value for each variable involved in the sequence. Specifically, the sequence contains three predicates showing the boundary of each variable. For example, the first predicate suggests the variable a is bigger than 0. In other words, variable a can be any value bigger than 0. Therefore, the method assigns an initial value to every variable according to predicates randomly. In this case, we suppose that the method assigns *a = 5, b = -10, c = 0*.

Second, the method will adjust these values to satisfy the formulation shown in the text box. The method adjusts the left part of the equation in priority. In particular, if the value of c is bigger or smaller the value of the right part, the method will decrease or increase the value of c until satisfying the equation. When our method tries all values in a boundary of variable c and the equation still does not meet our satisfaction, the method will change the value of the right part. However, change values of variables could affect other equations. Thus, the method uses a stack to record works. Specifically, if the method changes a value of a variable that can affect other equations, the method will push affected equations to a stack. Thus, if the stack is empty, all equations are satisfied. In this case, the method first attempt to change the value of c, but c exactly equal to 0. Thus the method attempt to affected equations changes the value of b. Because b - a equals to -15 lower than c, the method will increase the value of b until satisfying the equation. Finally, a = 5, b =5, c =0.

After the method generates all values and satisfies all equations. The method will generate a test case for the current sequence. The input part of the test case is the values of variables. The output part is the sequence. Thus, in this case, the test case is input: *a= 5, b = 5, c = 0* output: *start->T12->T24->T45->end*.

Sometimes background is merged into motivation, and is not required separately.

3. EXPERIMENT AND EVALUATION

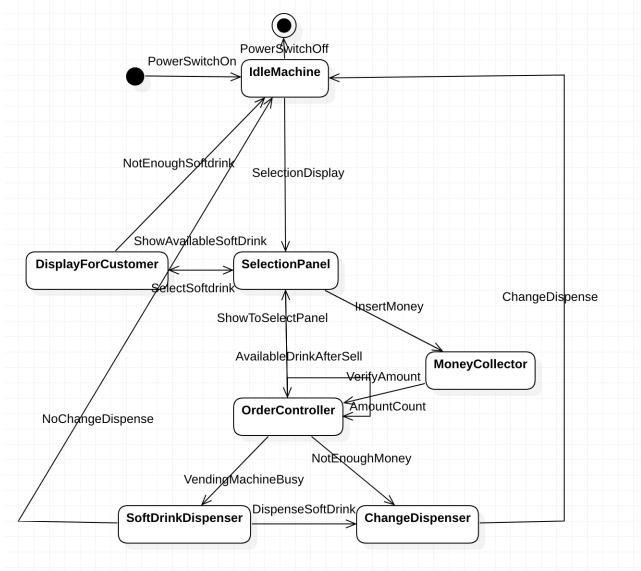

Figure 5: A vending machine statechart

3.1 Experiment

An experiment with thorough JUnit test cases and service implementation was applied to our algorithm. As is shown in **Figure 5**, a simple vending machine service was implemented to test the performance of our algorithm. The vending machine works as a core business logic for the system. As is shown in **Figure 6**, a data constraint file `DataInput.txt` was used specifically for this statechart to specific the transition sequence and generate test cases.

To better evaluate how our algorithm performs, two extra parameters were introduced here. Firstly, PathStatus records all the paths, which also represent transitions, that a business logic goes through. A PathStatus contains a list of string which represents the name of the transaction. This is a significant part to evaluate our work for that a thorough path coverage test case will be generated by our algorithm. And trying to evaluate the performance of this algorithm requires us to compare the actual path that a business logic executes to the expected path that our algorithm generated. `ReturnMoney` is the other parameter that records the money this vending machine is supposed to give back. `ReturnMoney` is designed specifically for the vending machine itself while PathStatus can be used in different experiments.

There are mainly two steps for evaluating our algorithm. The first step is to generate test cases. A VendingMachine.xmi file related to the statechart in **Figure 5** and a `DataInput.txt` file were used as the input for our experiment. After parsing the statechart into our custom defined

```
Predicates:
SelectSoftdrink:N <= 10
SelectSoftdrink:N > 0
NotEnoughSoftdrink:N > 10
InsertMoney:Amt >= 0
VendingMachineBusy:ReturnMoney >= 0
DispenseSoftDrink:ReturnMoney > 0
NotEnoughMoney:ReturnMoney < 0
ShowAvailableSoftDrink:P >= 3
ShowAvailableSoftDrink:P <= 7
AmountCount:TotalMoney > 0
VerifyAmount:Amt > 25
AvailableDrinkAfterSell:N >= 5
AvailableDrinkAfterSell:N <= 10
NoChangeDispense:ReturnMoney = 0
Formulation:
ReturnMoney = Amt - TotalMoney
TotalMoney = N * P
Order:
SelectSoftdrink InsertMoney
InsertMoney VerifyAmount
ShowToSelectPanel InsertMoney
ShowAvailableSoftDrink AvailableDrinkAfterSell
PathSelection:
VerifyAmount:Amt < 25
VerifyAmount:Amt > 0
AvailableDrinkAfterSell:N < 5
AvailableDrinkAfterSell:N > 0
```

Figure 6: Data input constraint file regarding the vending machine

data structure, this statechart was set into a state machine to exhaustively search for all the paths and generate all the transition sequences. Then, a filter was applied to the transition sequences to remove redundant and invalid paths. Finally, test cases were generated based on those valid paths and data was stored in Testcase class. The second step is to apply the test cases to our vending machine service and compare the result. Here we used JUnit to test and we matched return money and path records from our implementation of the vending machine and the result we got from the first step.

A pass to the test case that we generate indicates the implementation of our service is perfect regarding the path it goes through and the return money it gets back. A failure implies that for this particular test case, our implementation of the vending machine does not execute the path or return a certain amount of money as we expected. In order to evaluate our algorithm which mainly emphasizes the correctness of test case generation part. An assumption was made in our experiment that the implementation is unconditionally accurate. Thus, our implementation of the vending machine is a prototype that always executes precisely as long as we provide correct input data.

3.2 Result

84 distinctive transition sequences were generated after executing the state machine. Among all the sequences, 14 transition sequences were verified to be valid and able to generate correct input data as expected. **Table 1** shows the data input that we generated.

```
@Test
public void availableVerifyEnoughPurchase(){
    resultMsg = test( index: 13);
    Assert.assertEquals(resultMsg.getPathStatus().toString(),order.getSequence().toString());
    Assert.assertEquals(resultMsg.getReturnMoney(), order.getReturnMoney());
}

private ResultMsg test(int index){
    order = orders.get(index);
    VendingMachineService vendingMachineService = new VendingMachineImpl();
    return vendingMachineService.purchasing(order);
}

@After
public void printPath(){
    System.out.println("=============================================================================");
    System.out.println("PARAMETERS    : " + order);
    System.out.println("ACTUAL PATH   : "+resultMsg.getPathStatus());
    System.out.println("EXPECTED PATH : " + order.getSequence().toString());
}
```

Figure 7: An example of the JUnit test case generated from input data and how we perform a test on it

Table 1. Input data generated for vending machine service

Testcase\Input	OrderEmpty	Price	Number	InputMoney	ReturnMoney
NoInputDatat	true	0	0	0	0
NotEnoughDrink	false	4	58	-1	-1
NoChangeDispense	false	7	1	7	0
NotEnoughMoneyNotVerify	false	4	2	7	7
NotVerifyAmountPurchase	false	6	3	19	1
VerifyAmountNoChangePurchase	false	7	4	28	0
NotEnoughMoneyVerify	false	7	4	27	27
RegularPurchaseVerifyAmountChange	false	5	3	27	12
AvailableNoChangeNoVerify	false	3	8	24	0
AvailableNoVerifyNotEnoughMoney	false	7	5	18	18
AvailableNoVerifyChangePurchase	false	3	7	22	1
AvailableVerifyNoChangePurchase	false	7	7	49	0
AvailableVerifyNotEnoughMoney	false	7	5	34	34
AvailableVerifyEnoughPurchase	false	4	8	186	154

Each of the input data for the vending machine represents a unique transition sequence. By executing the vending machine service with one specific data input, a corresponding path record will also be generated. Take *"NotEnoughDrink"* as an example, the corresponding path records for this test would be *"PowerSwitchOn -> SelectionDisplay -> ShowAvailableSoftDrink -> NotEnoughSoftdrink -> PowerSwitchOff"*. This represents the situation that the number of drinks user has ordered exceeds the maximum limit for this vending machine. Another example would be *"AvailableVerifyEnoughPurchase"*, the transition sequence for this path is *"PowerSwitchOn -> SelectionDisplay -> ShowAvailableSoftDrink -> SelectSoftdrink -> AvailableDrinkAfterSell -> ShowToSelectPanel -> InsertMoney -> AmountCount -> VerifyAmount -> VendingMachineBusy -> DispenseSoftDrink -> ChangeDispense -> PowerSwitchOff"*. This path speaks for the situation that the number of drinks the user is trying to purchase reached the checking point. Also, a large amount of input money needs to be verified. Finally, money left after purchasing drinks needs to be returned.

These 14 distinct test cases are performed in our JUnit tests for this experiment. As we mentioned above, we assumed the implementation of the vending machine is precisely correct. So that after executing a test case, the vending machine will always act as we expected. As it is shown in **Figure 7**, orders list were generated in @BeforeClass annotation of our JUnit test. Our test cases were manu-

ally named and coded into the test. A correct implementation of the service we are testing will always pass the test cases. Meanwhile, a failure notifies us that there is something wrong with the service implementation. This property can be utilized as guidance when a developer is programming, which also serves TDD(Test Driven Development) process.

3.3 Evaluation

As we can see from the result of our experiment, test cases were generated as we expected. This not only helps with the implementation of a service but also assists developers to utilize it as the verification and validation tool to evaluate coding quality. Developers and quality engineers can both benefit from this process by unifying the inconsistency existing between these two communities. As we mentioned above, the result of our method can also serve for TDD, which turns requirements into test cases and shorten the developing cycle.

The threats to evaluation would be the assumption that a statechart with a corresponding text file provided by developers is always precise and error-free. Also, the constraint data input file only satisfies the single parameter situation, which is not enough for a real-life scenario. Meanwhile, in our experiment, we implemented the service specifically for our algorithm. That said, a false positive may exist in our experiment. Some real-life scenarios may have more complicated business logic and stateful processing of a data input. Under those circumstances, our test cases generated may not meet the requirement. However, by not automatically generating all the test cases by machine, manually coding test cases by software testing engineers can also benefit by taking our results as a reference.

4. RELATED WORK

The analysis of test cases generation is a hot topic these days. There is a lot of research regarding automatically generating test cases using UML diagrams. Our intention for researching on this topic is based on Automated-generating test case using UML statechart diagram [1]. This paper defined a methodology to generate test cases from UML state chart. However, their methodology only constructs simple user scenarios and is not able to configure flexible. What we have done in our algorithm is to provide users with another constraint file to configure the statechart as they want. Comparing to their results, our methodology is more user-friendly and scalable.

5. FURTURE WORK

The test case generation method based on statechart is definitely a favorable method to generate test cases for business system automatically. Our method can generate reasonable test cases for the business logic process of systems. However, the proposed method cannot manipulate dynamic variables changing with transition cycles in a statechart. For instance, in the vending machine statechart proposed in the experiment part, when users select a drink, the number of drinks will plus one. In other words, the values of variables can be changed with a different number of transitions. Further, we assume that all input state charts and constraints are correct and can be satisfied. Thus, a state chart validation function is a viable function to solve this problem. Another future work we consider is to develop an analysis function to show path coverage and errors of the given statecharts. Also, automatically generating test cases might be performed in the next version. This is also our intention for this project at the first stage while we were not able to do that in the current version.

6. CONCLUSIONS

In this paper, we developed an algorithm to generate test cases for a specific statechart with a corresponding data input constraint file. Through the experiment that we performed, the test cases generated by our algorithm did improve the efficiency of both developers and software testing engineers. A path coverage and boundary test analysis can also be generated through our method. While we do have some limitation with respect to the complexity of state transferring condition and the assumption of the correctness for the statechart user provided. This does not overwhelm the fact that our method of generating test cases facilitate the process of Test Driven Development and can be used for further improvement and analysis.

7. REFERENCES

[1] S. Kansomkeat and W. Rivepiboon. Automated-generating test case using uml statechart diagrams. In *Proceedings of the 2003 annual research conference of the South African Institute of computer scientists and information technologists on Enablement through technology*, pages 296–300, 2003.