2011 IEEE 13th International Symposium on High-Assurance Systems Engineering

Improve the Effectiveness of Test Case Generation on EFSM via Automatic Path Feasibility Analysis*

Rui Yang[1,2,3], Zhenyu Chen[1], Baowen Xu[1,2,+],W. Eric Wong[4], Jie Zhang[1]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
[2] Computer Science and Technology, Nanjing University, Nanjing 210093, China
[3] School of Electronic Engineering, Huaihai Institute of Technology, Lianyungang, 222005, China
[4] Department of Computer Science, University of Texas at Dallas, Richardson, TX 75803, USA
[+] Corresponding author: bwxu@nju.edu.cn

***Abstract*—A typical approach utilized for automated test case generation is to create a model of the implementation under test. Extended Finite State Machine (EFSM) is among the most popular models for model-based testing. However, automated test case generation on EFSM models is still a challenge task as a result of the fact that an EFSM model may contain infeasible paths. In this article we present a novel approach that combines static analysis and dynamic analysis techniques to address the problems of path infeasibility in the process of test case generation on EFSM models. A metric is presented for the purpose of finding a path subset that has few paths, long path length and goodness feasibility to meet adequacy coverage criteria. In addition, we develop an executable model to obtain run-time information feedback and introduce the Scatter Search into test case generation. Based on the executable model, the expected outputs associated with test data are also collected for construction of test oracles automatically. The experimental results show that our approach has good effectiveness for test case generation on EFSM models, and the method that combines static analysis and dynamic analysis can speed up the process of test case generation greatly.**

Keywords- test case generation, EFSM model-based testing, executable model, path feasibility analysis, test oracle

I. INTRODUCTION

A typical approach utilized for automated test case generation is to create a model of the implementation and use the model to generate test case [1]. Finite State Machine (FSM) and Extended Finite State Machine (EFSM) are most popular models for the purpose of model-based testing. However, FSM model can only represent the control part of a system while Extended Finite State Machine (EFSM) Model is an enhanced model of FSM that consists of states, variables and predicate among transitions can express both control flow and data flow of system. Unfortunately, although lots of automated test case generation methods for FSM models have been developed, test case generation for the EFSM models is still a challenging work.

In general, one of the challenges of automated test case generation on EFSM is the infeasible path problem, due to correlation of some transitions. The detection of an infeasible path is generally undecidable [2]. To the best of our knowledge, this is the first study to propose an approach to detect a large number of infeasible paths in the EFSM automatically. Moreover, generating test data to cover a given feasible path in an EFSM model is another challenging task [3]. Although some research has been done in the field of model-based testing, automated test data generation on an EFSM model is far from mature. Additionally, little attention has been paid to consider the automated test oracle creation for an EFSM model.

In this article, we present a novel approach for test case generation which takes the opportunity to bypass infeasible paths and meet adequacy coverage criterion as much as possible. The approach combines static analysis and dynamic analysis techniques to cover all transitions of the model with less paths which have a good probability of being feasible. Additionally, we develop an executable model such that run-time information feedback and the Scatter Search algorithm can be utilized to guide the test data generation process directly. Another advantage of using an executable model is that the test oracle can be created automatically.

In the first step, we generate a candidate set of paths with loops (including self-loops) from the EFSM model and present a metric through static analysis to evaluate the feasibility of a path and transition coverage of the model. Then, paths in the candidate set are sorted by the evaluation value and some infeasible paths identified are removed to facilitate test data generation in the next step. However, static analysis may be too coarse to detect infeasible paths in some cases. Thus, a dynamic analysis method, based on meta-heuristic techniques, is utilized to detect infeasible paths during the test data generation.

In the second step, we develop an executable model by means of semantic analysis of expressions, therefore the run-time information feedback technique and scatter search algorithm can be utilized to guide the test case generation process directly. Furthermore, the expected outputs associated with test data are also collected to construct the test oracle automatically. Finally, test data and test oracles are combined into complete test cases. In order to achieve specified coverage criterion, a subset of paths which generated test cases successfully and met specified coverage criterion are selected from the candidate set of paths.

The main contributions of this article are embodied in the following three aspects:

1. Static analysis and dynamic analysis techniques are combined to avoid infeasible paths as much as possible in the test case generation process and to satisfy test coverage criteria.

2. We develop an executable EFSM model via semantic analysis of expressions to support run-time information feed-back to generate test cases. Test oracles associated with the

*The work described in this article was partially supported by the National Natural Science Foundation of China (90818027, 60803007, 61170067) and the Fundamental Research Funds for the Central Universities.

1530-2059/11 $26.00 © 2011 IEEE
DOI 10.1109/HASE.2011.12

test data could be generated automatically, such that complete test cases are generated.

3. An experiment is designed and implemented to evaluate our approach. The experimental results confirm that our approach can speed up the process of test case generation greatly.

The remainder of this article is organized as follows: Section II introduces the basic concept of an EFSM model; Section III presents our approach of test case generation in detail; the experiments are described in Section IV; Section V reviews related work; Section VI concludes this article and provides some possible opportunities for future research.

II. PRELIMINARIES

An EFSM is a 6-tuple M=(S, s_0, V, I, O, T) where S is a finite states set of a model, $s_0 \in S$ is the initial state of a model, V is finite set of internal variables, I is set of input declarations, O is set of output declarations and T is finite set of transitions. Each transition $t \in T$ is also a 6-tuple: T= (s_i, s_j, a_t, o_t, P_t, A_t) where s_i is start state of t, s_j is end state of t, a_t is an input symbol where $a_t \in I$, o_t is output symbols where $o_t \in O$, P_t is a predicate operations on the current variable values and A_t is a sequential operations, e.g. output or assignment statements. P_t and A_t are also known as Guard and Action, respectively.

At the beginning, the model is at an initial state $s_0 \in S$ with initial variable values x_0. On the assumption that the machine is at the state s_i with the current variable values x_i. If model receiving input a_t or input event, and x_i is valid for P_t, e.g. $P_t(x_i)$=true, then M triggers the transition t and moves to the state s_j. In the process of transition, current variable values are changed by action A_t, and output variable values or output events may also be produced. Some transitions may have no associated predicate conditions, while some may have complex conditions that are difficult to be satisfied and some transition paths may be infeasible. The presence of such infeasible transition paths creates difficulties for automated test case generation for an EFSM model. In addition, some models contain both a start state and exit states, while some models are free of exit states.

An EFSM is deterministic if for any group of transition has the same input that leave a state, it is impossible to satisfy the guards of more than one transition at the same time in this transition group[4], otherwise the EFSM model is non-deterministic. In this article, we assume that the EFSM representation of the specification is deterministic and that the initial state is always reachable from any state through Reset operation.

III. OUR APPROACH

3.1 Candidate Path Set Generation Algorithm

The process of our approach to generate test cases from an EFSM can be split into two steps: first is to generate a candidate set of paths with loops (including self-loops) from the initial state on an EFSM model in order to satisfy the test coverage criteria and then produce test data for selected paths with the highest likelihood of feasibility. However, the generated paths may be infinite since it is possible to contain infinite loops. Consequently, we import a constraint condition that just contains loops or self-loops only one time to generate paths by means of our candidate paths set generation algorithm. This algorithm can also conveniently expand to generate paths that contain a certain number of loops (self-loops).

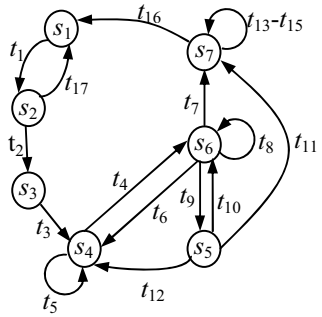

Figure 1.INRES protocol EFSM model

For the complexity of an EFSM model, in some cases more than two transitions associate with the same start state and end state and these transitions may be event-driven, in contrast to code-based testing where only the true and false branch of a node are verified, which is the reason for generating paths in advance for path-oriented testing on an EFSM model. For example, in the **Figure 1**, there are three self-loop transitions (t_{13}, t_{14}, t_{15}) from start state s_7 to end state s_7 (start state and end state can also different). Hence, it is necessary to transform state paths into transition paths by a combination algorithm.

Definition 1: A state path(SP) on an EFSM model is a sequence of states, that is $s_1 s_2 \ldots s_n$, such that there are transitions from state s_i to s_{i+1}, where $1 \leq i \leq n$-1.

Definition 2: A transition path (TP) on an EFSM model is a sequence of transitions $t_1\ t_2 \ldots\ t_n$, where every transition starts from the state that the previous transition finished.

We use the initial state of the EFSM model as the initial state of all generated paths for testing since the EFSM model is in the start state and values of the internal variables after a simple reset. The algorithm first traverses the EFSM model and in fact finds a simple path from the initial state to the other state and identifies the corresponding simple loops. A simple path is a path in a graph of an EFSM model which does not have repeating states. Similarly, a loop with no repeated states or transitions aside from the necessary repetition of the start and end state is a simple loop. After that, simple loops are combined into SP which contains the start state of simple loops, and every simple loop is utilized in a combination one time for each SP to generate SPs with loops. In order to facilitate the illustration, we denote this method as *Path-Gen,* and depict it in **Figure 2**.

In the *Path-Gen* algorithm, two type paths are generated, one type is SP which stored in the *SP_Result*, *SP_L_Result* and *P*, another type is TP which stored in the *TPS*. SPs consist of states (e.g. $s_1s_2s_3s_4s_6s_5$; in **Figure 1**) where TPs consist of transitions (e.g. $t_1t_2t_3t_4t_9$; in **Figure 1**). Firstly, the algorithm calls function *Getall_SP(s_0,s_k)* to get simple state paths from s_0 to s_k and stores in the *SP_Result*.

```
1:Algorithm Path_Gen
2: Input: store structure of EFSM model, roadList; state set S of EFSM
where s0 is initial state;
3: Output:state paths set contains all SP, SPS; transition paths set
contains all TP, TPS;
4: Goal:generate all transitions paths with one time loops from s0 to
other states;
5: backList:= Ø;    //store the states have been traversed
6: SP_Result:= Ø;   // store the simple paths
7: SP_L_Result:= Ø; //store the simple loop paths
8: CirList:= Ø;     //store the transitions which have loops
9: for each state sk in S
10: { Getall_SP(s0,sk);
11:   for (each start-end state pairs t in CirList)
12:       //si is the start state of t where sj is end state, each t in CirList
13:       // indicate that there exists simple loops from sj to si
14:      SP_L_Result ←Getall_sp(sj,si);
15:   for (each path spx in sp_L_Result)
16:      for (each path py in SP_Result)
17:         if py contains initial state of spx
18:            SPS←insert spx into py before initial state of spx;
19:   Transform state paths set SPS into transition path set TPS;
20:  }

21: Getall_SP(start_node,destination_node)
22: { add start_node to backList;
23:  for (each transition t in roadList)
24:      //sj is the start state of t where sj is end state
25:     { if (si == strat_node)
26:        if (sj==destination_node)
27:           SP_Result←concatenate backList and destination_node;
28:        else
29:           if (backList not contains sj)
30:              Getall_SP(sj,destination_node);
31:           else
32:              CirList← start-end state pairs of simple loops;
33:     }
34:   Remove start_node from backList;
35:  }
```

Figure 2. Pseudo code of the candidate path set generation algorithm

In the above process, start-end state pairs of simple loops (see line 32, **Figure 2**), that indicate the presence of loop paths from end state to start state, are identified and stored in *CirList*. Then, each start-end state pair that is stored in *CirList* is chosen to generate simple state paths from end state to start state as simple state loop paths. After that, simple state loop paths are combined into a simple path which contains the start state of simple loops (see lines 15-18, **Figure 2**). Finally, we developed a combination algorithm to expand multi-transition to transition path respectively, all the state paths with loops in *SPS* are transformed into transition paths with loops in *TPS* (see line 19, **Figure 2**).The details of the combination algorithm are not covered in this article due to limitation of space.

3.2 Feasibility Evaluation Strategy

A challenging problem of path-oriented test case generation on EFSM model is the existence of infeasible paths, that is to say there is no input data to traverse these infeasible paths. Unfortunately, the detection of these infeasible paths is generally undecidable[2]. The main cause of existing infeasible paths is due to the variable correlation among the actions and predicate conditions. Therefore exploring the correlation between actions and predicate conditions plays an important role in detecting infeasible paths. If a majority of infeasible paths can be detected on the *TPS* in advance, it will greatly improve the performance of test case generation by means of avoidance of infeasible paths. To detect the feasibility of paths that generated above, we extend Kalaji's dependencies analysis values[3] and present a metric to evaluate the feasibility of a path. Furthermore, static data flow analysis technique is used to identify infeasible paths directly. Before giving a detailed description of our approach, we introduce some technical definitions and terms:

Definition 3: A TP is said to be a feasible transition path (FTP) if and only if it is possible to traverse each transition t_i in the sequential order in TP , where $1 \le i \le n$.

In general, the guard (predicate) of a transition has the form ($e\ g^{op}\ e$') where e and e' are expressions and g^{op} is the predicate operator(e.g.<, >, ≠, =, ≤, ≥). Given an expression e, *Ref(e)* denote the set of variables in this expression. According to e and e' , The guard of a transition can be classified into the three types as follows:

1. g^{pv}: g^{pv} is a comparison operation between a parameter and zero or more context variables; there exists $p \in Ref(e) \cup Ref(e')$, where p is a parameter.
2. g^{vv}: g^{vv} is a comparison operation among values of context variables; every element of *Ref(e) Ref(e')* is a context variable.
3. g^{vc}: g^{vc} is a comparison operation between a constant variable and an expression involving values of context variables; all elements of $Ref(e) \cup Ref(e')$ are context variables, where e or e' is a constant.

An assignment operation that occurs in a transition t has the form of $v=e$, where v is a context variable , e is an expression. An assignment to a context variable v can be classified into the types as follows:

1. op^{pv}: op^{pv} assigns to v a value that depends on the parameter hence parameter $p \in Ref(e)$.
2. op^{vv}: op^{vv} assigns to v a value that depends only on one or more context variables hence all the elements of *Ref(e)* are context variables.
3. op^{vc}: op^{vc} assigns a constant value to v hence e is a constant.

Definition 4:There exists a definition-p-use pair transition $<t_i , t_j>$ on a TP iff $t_i \in$TP has an assign statement to variable v and $t_j \in$TP has a predicate logic(guard) use on the variable v and the path from t_i to t_j not exist reassignment of v, where $1 < i < j < n$,n is path length.

In order to achieve our objective, metric that evaluates the feasibility of a transition path is required. There are some factors that need to be considered when assigning a penalty value to definition-p-use transition pairs on a path. Firstly, it is always able to take a transition without predicate condition since no guards need to be satisfied. It has been known that comparison operator ≠ is easiest to be satisfied while = is the most difficult. Similarly, the guard type that occurs in definition-p-use transition pairs determines the degree that a predicate use transition can be impacted by a transition with variable definition.

TABLE I. THE SUGGESTED PENALTY VALUES OF DEFINITION-P-USE PAIR

Guard	Action		
	op^{pv}	op^{vv}	op^{vc}
$g^{pv}(=)$	8	16	24
$g^{pv}(<,>)$	6	12	18
$g^{pv}(\leq,\geq)$	4	8	12
$g^{pv}(\neq)$	2	4	6
$g^{vv}(=)$	20	40	60
$g^{vv}(<,>)$	16	32	48
$g^{vv}(\leq,\geq)$	12	24	36
$g^{pv}(\neq)$	8	16	24
$g^{vc}(=)$	30	60	If Case 1 infeasible else 0
$g^{vc}(<,>)$	24	48	If Case 3,Case 2 infeasible else 0
$g^{vc}(\leq,\geq)$	18	36	If Case 5,Case 4 infeasible else 0
$g^{vc}(\neq)$	12	24	If Case 6 infeasible else 0

For example, g^{pv} is the guard type that is easiest to be satisfied since the parameter value can be assigned to satisfy this guard. On the contrary, g^{vc} is the guard that is hardest to be satisfied. Furthermore, the assignment type of a definition transition should influence the complexity of a predicate use when a parameter value is involved but has a negative effect, especially when a constant value is involved. For formalization, we present Property as follows:

Property 1: Let $<t_i, t_j>$, a definition-p-use pair on a path TP, t_i assign statement to variable v with respect to constant value C_1 and $t_j \in$TP has a predicate use on the variable v with respect to constant value C_2, then the paths are infeasible in the following cases:

Case 1: Where guard operator of g^{vc} in t_j is "=", if the constant variable C_1 is different from C_2.

Case 2: Where guard operator of g^{vc} in t_j is ">", if the constant variable $C_1 \leq C_2$.

Case 3: Where guard operator of g^{vc} in t_j is "<", if the constant variable $C_1 \geq C_2$.

Case 4: Where guard operator of g^{vc} in t_j is "≥", if the constant variable $C_1 < C_2$.

Case 5: Where guard operator of g^{vc} in t_j is "≤", if the constant variable $C_1 > C_2$.

Case 6: Where guard operator of g^{vc} in t_j is "≠", if the constant variable C_1 is same to C_2.

This property can be proved by reduction to absurdity and Case 2 is proved as follows. The other cases can also be proved using the same methods.

Proof: In the case 2, v is assigned constant value C_1 in the form of $v=C_1$ in transition t_i, t_j has the predicate use $v>C_2$, if result of $v>C_2$ is true, it can deduced that $C_1>C_2$; this contradicts with the original hypothesis that $C_1 \leq C_2$; therefore result of $v>C_2$ is false, that means the path containing definition-p-use pair $<t_i, t_j>$ is infeasible.

Beside the aforementioned cases, the suggested penalty values for evaluating the feasibility of a transition path are shown in **TABLE I**.

When there is more than one condition in a guard of transition, the penalty value associated with this guard rest with the logical operator among these conditions. In these cases, we consider the "*AND*" operator with the sum of the penalties while the "*OR*" operator only the lowest penalty is used.

There are several test strategies for state-based model such as state coverage, all-transitions coverage, one-loop-paths coverage and all-paths coverage that require the generation of a set of FTPs in order to produce a test suite[5]. However, a stronger coverage criterion often requires a lot more test cases to satisfy it. In addition, from previous studies, it is shown that the covering array test case improves fault-detection efficiency with small number of longer test cases compared to shorter ones. Hence, for the purpose of finding a subset of path that has fewer paths, longer path length, goodness of feasibility and ease of triggering to cover all transitions of EFSM model from *TPS*, we present the metric as follows:

$$f = \begin{cases} \dfrac{\sum_{i=0}^{k} v(df_i)}{|TP|^d} & If \quad \sum_{i=0}^{k} v(df_i) \neq 0 \\ 0 - |TP| & If \quad \sum_{i=0}^{k} v(df_i) = 0 \end{cases} \qquad (1)$$

where df_i is the definition-p-use transition pair of a path, $v(df_i)$ is the penalty value of definition-p-use transition pair that listed in TABLE I and k is the number of definition-p-use transition pairs in a path. In addition, $|TP|$ is the length of a path, d is a value that is used to adjust weight of path length in the metric. For the reason that a longer path has higher probability that it contains more definition-p-use transition pairs that can result in larger penalty value, denominator $|TP|$ is the trade-off between path length and penalty value. However, if the sum of the penalty value of definition-p-use transition pair on a path equals zero ($\sum_{i=0}^{k} v(df_i) = 0$) it means the path has best feasible probability because no definition-p-use interdependence exists among transitions. In this case, a longer path should get a lower penalty value to cover more transitions of an EFSM model, therefore the penalty value is assigned $0-|TP|$. This metric also is also easy to be expanded for other test coverage strategies.

The above formula is applied to evaluate the value of each path in the *TPS*, in order to put the formulas into implementation, we take advantage of a backward traverse algorithm on each path to identify definition-p-use pairs to compute penalty value f. In the process of evaluation, the paths which contain incompatible definition-p-use pair according to one of the six cases in **Property 1** are considered infeasible hence it will be removed from the *TPS*. After that, the paths in *TPS* are sorted by f in ascending order. However, a subset of feasible path which cover all-transitions or other coverage strategies will be generated dynamically in the process of test case generation.

3.3 Test Case Generation Algorithm

Although the metric to predict the feasibility of a path is proposed in the foregoing section, it is still a proximity detecting method that cannot predict the infeasible path thoroughly. Consequently, we proposed a dynamic analysis method to detect infeasible paths, which is based on meta-heuristic search to detect infeasible paths during the test data

generation. In fact, dynamic techniques cannot distinguish the feasibility of paths directly. In dynamic techniques, a common strategy is to limit the number and the depth of search. That is to say, if the input sequences that traverses a target path has not been found in the final period of the search, the path will be considered infeasible[6].

Figure 3.Pseudocode of test case generation on executable EFSM models

```
1:Algorithm TS_Gen
2:Input: sorted TPS;
3:Outputs: test suit Resultset and its corresponding feasible paths subset
of TPS;
4: Corresponding outputs set O_p when generating test cases.
5:for each path p_t in sorted TPS
6: {
7:   if (all transitions of model are covered) exit;
8:   if (p_t contains transitions not be covered)
9:      { Tempset←SS_Path_TC(p_t);
10:       if (Tempset covered p_t)
11:         { add Tempset into Resultset and record p_t ;
12:           update transition coverage information; }}
13:  else
14:      skip and get next path;
15: }

16:Solution set SS_Path_TC(transition path p_t)
17:{   Generate an initial improved test cases Solution set P for p_t;
18:    while (iteration times<max iteration times)
19:    {  Apply populations in P to executable model;
20:       Refset ← m best solution of P set;
21:       if (all paths value equals 100)
22:          { O_p←outputs of executing populations;
23:            return Refset;
24:          }
25:       while (B ← Combine solutions from RefSet and B≠Ø)
26:          { Improve solution in B;
27:            Apply populations in Refset to executable model;
28:            RefSet ←keep m best from RefSet ∪B;
29:            if (all paths value equals 100)
30:              { O_p←outputs of executing populations;
31:                return Refset;
32:              }
33:          }
34:       clear P set;
35:       P set← RefSet;
36:       fill the P set with diverse solutions;
37:    }
38:    return Refset;
39:}
```

3.3.1 Executable EFSM model

An executable model is a model that is complete enough to be executed to simulate the behavior of the implementation system in real world, it can be tested as a prototype rather than an abstract diagram. In contrast to common static models, an executable model defines the dynamic behavior as part of the model. Through the use of execution semantics, executable model has the ability of models to be directly executable.

To make a static model executable, semantic analysis of expressions becomes the key issue to construct the executable model. When conditional expressions and action expressions on transitions of the EFSM model are parsed, the valid test cases involving input parameters, context variables and events can apply to the expressions on transitions of the path and acquire the outputs of transitions, hence an executable model has become similar to an executable program. The validity of test cases can also be verified by monitor executing trace during the test data generation. To achieve this goal, we utilize the excellent open java library JEval[7] to parse and evaluate dynamic and static expressions of an EFSM model. JEval can be used to parse mathematical, Boolean, String and functional expressions at run time and it supports most operators. When the EFSM model is at a certain state with the variable values and received input values or input events, if guard statement is parsed and valid, then the model takes the transition and moves to the succeeding state; in the process of transition, action statements are also parsed by JEval and current variable values may be changed, or also produce output events.

3.3.2 Run-time Information Feedback Technique

Run-time Information feedback refers to information that is obtained during test execution. This technology was originally proposed by Miller and Spooner[8], their technique is applied to obtain execution feedback for code based testing. In our method, the initial test data is to be generated into a seed set randomly and executed on an executable EFSM model immediately. During its execution, feedback information obtained from the executable model is used to generate new, improved test cases.

The feedback information we collected in the process of applying a test case on a path of executable EFSM model to guide test generation is calculated as follows:

$$F_e = \frac{|SP_t|}{|TP|} \times 100 \qquad (2)$$

where $|TP|$ is the length of a path which is a specified path to generate test cases while $|SP_t|$ is the length of a sub-path that is triggered from source transition sequentially by the current test case. if F_e equals 100, that means the test case can trigger all transitions of the path, hence the test case is generated successfully.

3.3.3 Scatter Search Algorithm

Scatter search is a population based meta-heuristic algorithm that begins to generate initial diverse solutions set via using a diversity generation method, each solution is processed by the improvement method and the resultant individual is added to the initial set. Then the best solutions from the initial set be selected to create the reference set and the most diverse in relation to the solutions already in the reference set. In the next step, solutions in the reference set are grouped into subsets of two or more individuals using the subset generation method. After that, solutions in each subset are combined to produce new individuals[9], and this combination process is defined by the solution combination method. The improvement method is applied to each new generated solutions, a reference set update method evaluates the new solution through execution feedback information to verify whether they can update the reference set, as they have better fitness value(See formula 2) than some solutions

stored in the set. If so, the best solutions are included in the reference set and the worst solutions are dropped[10]. This loop is executed until a stopping condition is fulfilled or the final solution of the problem to solve is stored in the reference set.

The outline of the test case generation method that combines execution information feedback and scatter search template is shown in **Figure 3**.

Although the static analysis method has been proposed to evaluate the feasibility of a transition path and some incompatible paths are removed from *TPS*, it is still a proximity detecting method that cannot predict the infeasible path thoroughly. Therefore, a dynamic analysis technique is used in the process of test case generation. In the aforementioned algorithm *TS_Gen*, a path in *TPS* is picked up in sequence firstly to check whether the transitions of path have already been covered (see line 7, **Figure 3**); if so, this path is skipped and next path is chosen for the next iteration. Otherwise, test case generation process starts by means of Scatter Search template with aforementioned five steps (see lines 17-39, **Figure 3**). If test cases have not been found in the final period of the search, the path is also considered infeasible. The function *SS_Path_TC* is used to generate test cases for a specified path p_t, in the process of scatter search, run-time information (see **Formula 2**) is calculated to guide the search. If the value of F_e equals 100 that means test cases for the p_t are generated successfully and the corresponding outputs of executing those test cases are recorded into O_p to create the test oracle.

IV. EMPIRICAL STUDY

In this section we present a detailed empirical study with the goal of determining whether test case generation with specified coverage criterion for an EFSM model can benefit from our technique. More specifically, the study is designed to empirically answer the following questions:

Q1: What is the effectiveness of our approach for test cases generation? How many transitions are covered in the model? How many paths in *TPS* are executed for test cases generation, and how many paths are valid?

Q2: What is the effectiveness of static analysis and dynamic analysis techniques for the path feasibility?

Q3: What is the time efficiency of the approach?

To answer the above questions, an empirical study is conducted using four popular EFSM models: Automated Teller Machine (ATM)[11], INRES protocol[12], Class 2 transport protocol[13], and SCP protocol[14]. ATM contain both a start and exit state, others are free of exit states. For generality, we do not consider that the path in the model with exit state must end in an exit state in our experiment.

4.1 Study Procedure

In order to achieve the experiment, firstly we extract the details of aforementioned four EFSM models (states, transitions, guard statement and action statement) and store the structure into files, then, the file will be loaded by our program one after another to generate test cases. The main configuration of Scatter Search algorithm we set is listed as follows:

1. Test data Solution set *P* size=50;
2. *Refset* size=2;
3. Max iteration times=20000;
4. SBXCrossover probability =1.0;
5. SBXCrossover Distribution Index=20.0

Max iteration times refer to the total count of evaluation time for each new solution, which is generated in Diversification Generation, Subset Generation and Improvement steps in the *SS_Path_TC*. Since all-transitions coverage criterion is specified in this article, **Formula 1** is applied in the experiment and *d* is assigned 0.8 here.

Each test case starts with a source state in the EFSM model, the process ensures that the test suite satisfies the all-transitions coverage criterion (it is easy to expand to other model-based test criteria). For creating the test oracles automatically, outputs associated with each individual test case also recorded for these test cases. In addition, to address the **Q1**，we record some additional statistical data when the program is running. The results are listed in **TABLE II.**

State number and transition number is the property of the subject model, number of *TPS* is the count of paths generated by the *Path_Gen* algorithm, since there are more multi-transitions between same start state and same end state on ATM model and Class 2 protocol (e.g., self-loop s_4 to s_4 has 9 transitions in Class 2 model), number of TP on these two models is also relative larger. Number of sorted *TPS* is the count of paths after removing infeasible paths that were identified by static analysis. FTP number of the result set is the count of feasible transition paths that were selected in the process of test case generation to achieve all-transitions coverage. However, as shown in TABLE II, 100 percent transition coverage ratio is difficult to achieve for path-oriented testing on EFSM model, because of the existence of loops (self-loops), e.g. , in ATM model, transition t_3 cannot be covered because three times self-loop transition t_2 are need to trigger t_3, thus only TP $t_1t_2\ t_2\ t_2\ t_3$ can be triggered. *TPS* only contains $t_1\ t_2\ t_3$ since the path contains loops or self-loops only one time by means of our paths set generation algorithm. Although Chan-son et al.[15]made some efforts to determine how many times the loop should be repeated so that test cases become executable. This method is not appropriate for complicated EFSM models and cannot be used if the number of loop iterations is not known. Similar situation also occurs in other models, SCP model achieve lowest coverage ratio among the four EFSM models.

In order to illustrate the advantage of using static analysis in our method, we achieve the experiment of test case generation without static analysis technique. It is means that the *TS_Gen* algorithm is applied to generate the path set without the evaluation process, the infeasible paths identified by static analysis technique are kept and the path set is also not to be sorted. Since there are tiny differences in execution time of the test case generation of the model, for the purpose of illustrating the problem precisely, we run the program 30 times for every model and calculate its average value of execution time. The experiment results (see **Table III**) show that the method combining static analysis and dynamic

TABLE II. THE STATISTICS RESULT OF FOUR OBJECTIVE MODELS

Subject model	State number	Transition number	number of *TPS*	number of sorted *TPS*	input parameters	FTP number of result set	Covered transitions	Coverage ratio
ATM	9	23	349	172	6	16	22	95.6%
INRES protocol	8	18	109	109	2	10	18	100%
Class2 protocol	6	21	334	320	10	16	20	95.2%
SCP protocol	4	8	29	18	4	2	6	75%

TABLE III. EXPERIMENT RESULT COMPARISON BETWEEN TEST CASE GENERATION WITH STATIC ANALYSIS AND TEST CASE GENERATION WITHOUT STATIC ANALYSIS

Model& Method	Subject model	Executed path number	FTP number	FTP iteration times	Execute time(sec.)
Without static analysis technique	ATM	18	16	1600	8
	CLASS 2	44	16	2522	114
	INRES	11	10	11954	5
	SCP	22	3	1022	29
With static analysis technique	ATM	17	16	1600	4
	CLASS 2	30	16	2016	57
	INRES	10	10	7650	2
	SCP	10	2	816	13

analysis technique improved the performance greatly. In the SCP model, 12 paths are reduced and the valid paths achieved the same coverage ratio are also reduced; in the CLASS 2 model 14 paths are reduced. The correlation among transitions in the ATM and INRES models is relatively simple, therefore the executed path and valid path were reduced minimally; however, the execution time and the iteration times of feasible transition path (FTP) were reduced greatly. The experiment results also answer the **Q3**, the execution time of test case generation is completely acceptable for all-transition coverage of the model in our experiment configuration.

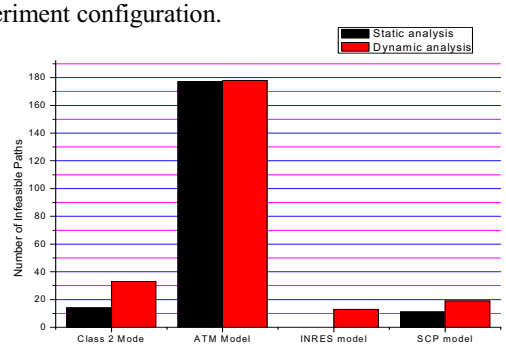

Figure 4.Comparison result of infeasible paths that were identified by static analysis and dynamic analysis

To address **Q2** and achieve further analysis of the results of **Table II**, we remodified the *TS_Gen* algorithm to apply for all paths in *TPS* (including infeasible path identified by static analysis). In this experiment we also run the program 30 times for every model. **Figure 4** shows the number of infeasible paths identified in test generation process contrasted with that of the static analysis process in our experiment. It shows that static analysis techniques can only detect a part of an infeasible path, and the structure of the EFSM model also affects the number of infeasible paths. For example, there are six infeasible definition-p-use pairs in INRES model whereas no infeasible paths are detected because all these definition-p-use pairs are not contained in one transition path when generating paths via *Path_Gen* algorithm. However, the method that combines static analysis technique and dynamic analysis technique can speed up the process of test case generation greatly.

V. RELATED WORK

Some methods have been proposed for the purpose of test sequence generation from EFSM models [12][16]. They focused on test sequence generation with data flow testing and control flow has been ignored or considered separately. Other methods that test from an EFSM model using FSM-based testing techniques [17][18]. However, the number of states in the resulting FSM can be subject to explosion. While there has been work on transforming an EFSM to one that has no infeasible paths [19][20], the approach converts a class of EFSM into consistent EFSM in order to enable test sequences generation. However, it has only been achieved for EFSMs in which all operations and predicate statements are linear and has exponential complexity. Moreover, the approach does not provide a technique to generate input test sequence that will test the generated paths.

Derderian et al. [21] give a fitness function that estimates the feasibility of a path. Kalaji et al. [22] introduced the approach of generating feasible transition paths that are easy to trigger with a genetic algorithm. However, adequacy coverage criterion has not been considered in the literature, and the path length should be determined in advance. Lefticaru et al. [23] proposed a fitness calculation method to derive test data from a state machine. The approach of fitness calculation is based on Tracey et al. research [24]. However, in some cases, this approach does not always provide a sufficient guidance.

Some meta-heuristic techniques have been used to generate test data; Genetic Algorithms are most popular. Another meta-heuristic technique that can be applied to automated test case generation is scatter search. However, the only articles that use the scatter search technique to automate the generation of test cases in [10][25], which use this technique to obtain branch coverage. The scatter search

algorithm which has been compared with Genetic Algorithms (GAs) in some problems, producing high quality solutions in fewer evaluations than GAs [26].

VI. Conclusions and Future Work

This article presented a novel approach to generate test cases on EFSM model for achieving adequacy coverage criterion. This approach integrates static analysis and dynamic analysis technique to address an important problem in test data generation for an EFSM model. Moreover, we developed an executable EFSM model via semantic analysis of expressions to support run-time information feedback technique to generate test cases. Depending on the advantage of using an executable model, automatic generation of oracle information becomes available. The experimental results show that our method has good effectiveness for test case generation on EFSM model, and the method that combines static analysis technique and dynamic analysis technique can speed up the process of test case generation greatly.

It should be noted that our experimental results are still preliminary. In our future study, we intend to utilize the multi-objective optimization to trade-off the contradiction among path length, penalty value of feasible evaluation and coverage ratio of a path. In addition, more static analysis technique would be used to deal with infeasible paths and improve the ratio of coverage criterion.

Reference

[1] X. Yuan and A. M. Memon, "Using GUI Run-Time State as Feedback to Generate Test Cases," *International Conference on Software Engineering (ICSE '07)*, 2007, pp. 396-405.

[2] D. Hedley and M. A. Hennell, "The Causes and Effects of Infeasible Paths in Computer Programs," *International Conference on Software Engineering (ICSE '85)*, 1985, pp. 259-266.

[3] A. S. Kalaji, R. M. Hierons, and S. Swift, "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)," *International Conference on Software Testing Verification and Validation (ICST '09)*, 2009, pp. 230-239.

[4] C.-hua Shih, J.-dar Huang, and J.-yang Jou, "Stimulus Generation for Interface Protocol Verification Using the Non-deterministic Extended Finite State Machine Model," *IEEE International High-Level Design Validation and Test Workshop*, 2005, pp. 87-93.

[5] L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader, "Requirement-based Automated Black-box Test Generation," *International Computer Software and Applications Conference (COMPSAC '01)*, 2001, pp. 489-495.

[6] S. Sai-ngern, C. Lursinsap, and P. Sophatsathit, "An Address Mapping Approach for Test Data Generation of Dynamic Linked Structures," *Information and Software Technology*, 2005, vol. 47, pp. 199-214.

[7] R. Breidecker, "http://jeval.sourceforge.net." .

[8] W. Miller and D. L. ; Spooner, "Automatic Generation of Floating-Point Test Data," *IEEE Transactions on Software Engineering*, 1976, vol. 2, no. 3, pp. 223-226.

[9] A. J. Nebro, F. Luna, E. Alba, B. Dorronsoro, J. J. Durillo, and A. Beham, "AbYSS: Adapting Scatter Search to Multi-objective Optimization," *IEEE Transactions on Evolutionary Computation*, 2008, vol. 12, no. 4, pp. 439-457.

[10] R. Blanco, J. Tuya, and B. Adensodiaz, "Automated Test Data Generation Using a Scatter Search Approach," *Information and Software Technology*, 2009, vol. 51, no. 4, pp. 708-720.

[11] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of State-based Models," *International Conference on Software Maintenance (ICSM '03), 2003*, pp. 34-43.

[12] C. M. Huang, M. Y. Jang, and Y. C. Lin, "Executable EFSM-based Data Flow and Control Flow Protocol Test Sequence Generation using Reachability Analysis," *Journal of the Chinese Institute of Engineers*, 1999, vol. 22, no. 5, pp. 593-615.

[13] T. Ramalingom, K. Thulasiraman, and A. Das, "Context Independent Unique State Identification Sequences for Testing Communication Protocols Modeled as Extended Finite State Machines," *Computer Communications*, 2003, vol. 26, no. 14, pp. 1622-1633.

[14] A. Cavalli, C. Gervy, and S. Prokopenko, "New Approaches for Passive Testing Using an Extended Finite State Machine Specification," *Information and Software Technology*, 2003, vol. 45, no. 12, pp. 837-852.

[15] S. T. Chanson and J. Zhu, "A Unified Approach to Protocol Test Sequence Generation," *The Conference on Computer Communications, Proceedings (INFOCOM '93)*, pp. 106-114, 1993.

[16] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing," *IEEE Transactions on Communication*, 1991, vol. 39, no. 4, pp. 514-523.

[17] A. Petrenko, "On Fault Coverage of Tests for Finite State Specifications," *Computer Networks and ISDN Systems*, 1996, vol. 29, no. 1, pp. 81-106.

[18] K.-T. Cheng and a S. Krishnakumar, "Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model," *ACM Transactions on Design Automation of Electronic Systems*, 1996, vol. 1, no. 1, pp. 57-79.

[19] M. Ü. Uyar and A. Y. Duale, "Test Generation for EFSM Models of Complex Army Protocols with Inconsistencies," *21st Century Military Communications. Architectures and Technologies for Information Superiority*, 2000, vol. 0, no. C, pp. 340-346.

[20] A. Y. Duale and M. Ü. Uyar, "A Method Enabling Feasible Conformance Test Sequence Generation for EFSM Models," *IEEE Transactions on Computers*, 2004, vol. 53, no. 5, pp. 614-627.

[21] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo, "Estimating the Feasibility of Transition Paths in Extended Finite State Machines," *Automated Software Engineering*, 2009, vol. 17, no. 1, pp. 33-56.

[22] A. S. Kalaji, R. M. Hierons, and S. Swift, "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine," *International Conference on Software Testing Verification and Validation (ICST '09)*, 2009, pp. 230-239.

[23] R. Lefticaru and F. Ipate, "Automatic State-Based Test Generation Using Genetic Algorithms," *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '07)*, 2007, pp. 188-195.

[24] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An Automated Framework for Structural Test Data Generation," *IEEE International Conference on Automated Software Engineering (ASE'98)*, 1998, pp. 285-288.

[25] R. Sagarna and J. Lozano, "Scatter Search in Software Testing, Comparison and Collaboration with Estimation of Distribution Algorithms," *European Journal of Operational Research*, 2006, vol. 169, no. 2, pp. 392-412.

[26] F. Glover, M. Laguna, and R. Martí, "Fundamentals of Scatter Search and Path Relinking," *Control and Cybernetics*, 2000, vol. 29, no. 3, pp. 653-684.