Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM) with the Counter Problem

Abdul Salam Kalaji, Robert Mark Hierons and Stephen Swift
School of Information Systems, Math and Computing, Brunel University, Uxbridge, UB83PH, UK
e-mail: {abdulsalam.kalaji,rob.hierons,stephen.swift}@brunel.ac.uk

Abstract— **The extended finite state machine (EFSM) is a powerful approach for modeling state-based systems. However, testing from EFSMs is complicated by the existence of infeasible paths. One important problem is the existence of a transition with a guard that references a counter variable whose value depends on previous transitions. The presence of such transitions in paths often leads to infeasible paths. This paper proposes a novel approach to bypass the counter problem. The proposed approach is evaluated by being used in a genetic algorithm to guide the search for feasible transition paths (FTPs).**

I. Introduction

Testing is an important part of the system development process. However, conventional manual testing is known to be expensive and imprecise and so there has been much interest in developing methods that can automate testing [1]. In this paper, we focus on conformance testing which aims to determine whether the implementation under test (IUT) agrees with its specification.

In order to conduct conformance testing, it is desirable to have a model from which test sequences can be derived. The EFSM is a modeling approach that is commonly used for representing state-based systems [2]. The EFSM consists of a finite set of states, transitions and context variables. Any transition can have guards and operations where the guards must be satisfied so that the transition can be fired and thus the associated operations are executed [2]. Although EFSMs are a popular modeling approach, automatic testing from an EFSM is complex due to the presence of data and guards (preconditions) which can result in infeasible paths. Testing from an EFSM can be based on coverage criteria such as transition coverage. When attempting to satisfy a criterion it is normal to produce a set of transition paths (TPs) through the EFSM that satisfy the criterion and then produce test data to trigger the paths. However, the problem of generating feasible transition paths (FTPs) from an EFSM is generally undecidable [3]. Furthermore, developing good methods to derive FTPs from an EFSM is an open research problem [4]. One important problem that can result in a TP being infeasible is the presence of a transition whose guard(s) references a counter variable. For example, if an EFSM has a counter variable c whose value is initially 0, a transition t_i that has a guard $c > 2$ cannot be exercised unless a transition t_j which updates the value of c has already occurred a sufficient number of times.

Although optimization algorithms, such as genetic algorithm (GA), have proven to be efficient in test automation [1], there has been very little work on using them in testing from an EFSM. Our previous work [5] proposed a TP fitness metric based on analyzing the data dependencies in a given TP in order to estimate its feasibility. The proposed TP fitness metric can be utilized in a search to generate a set of FTPs that are likely to be feasible. However, for an EFSM that contains a counter variable (i.e. a variable that counts how many times a transition is repeated), a given TP may include a transition that has a guard over the counter variable and so there is a need to determine which other transitions are involved (those that affect the counter variable value) and how many times they must be called. The counter problem thus requires additional analysis. Unfortunately, this is a substantial mathematical problem [3] and counter variables are a significant problem for search based testing (see for example [1, 6]).

In this paper we present a novel approach based on control and data analysis that extends the TP fitness metric of [5] to automatically determine whether a given TP includes a transition whose guard references a counter variable, which other transitions are involved (are required in this TP) and how many times they must be called.

The approach presented in this paper aims to form part of the solution to the following problem:
Given: a test adequacy criterion and an EFSM model that includes counter variables
Problem: generate a set of TPs that are feasible and satisfy the test criterion.

The primary contributions of this paper are the following:
1. It proposes a method to bypass the counter problem by automatically determining whether a transition guard references a counter, which other transitions are involved and how many times they have to be called
2. It shows that the proposed approach is effective in generating FTPs from EFSMs models with counter variables to satisfy the test criterion.

II. The Proposed FTPs Generation Approach

The TP fitness metric proposed in our previous work [5] penalizes the dependencies found among a TP's transitions in order to estimate the TP feasibility. Importantly, the fitness metric can be computed quickly and so is suitable for use in search-based testing. However, problems can occur when a given TP includes a transition guard that references a counter variable and in this case the search may not receive the necessary guidance. Here, the problem is that in order to execute a transition t' whose guard references a counter variable we must first execute another transition(s) t (that

978-0-7695-4050-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICSTW.2010.25

TABLE 1 & 2. GUARDS AND OPERATIONS REPRESENTATION

Guard	Representation
no guard	0
$=$	1
$>$	2
$<$	3
$\geq$	4
$\leq$	5
$\neq$	6

Operation	Representation
no operation	0
$op^{v=c}$	1
op^{v+c}	2
op^{v-c}	3
$op^{v\times c}$	4
$op^{v/c}$	5

updates the counter variable value) and a certain number of times so that the guard of t' is satisfied.

To overcome this problem, the test criterion should include the required extra transitions together with the number of times they have to be called. Generally, this problem is a challenging mathematical task [3]. However, in some cases it can be approached with an abstraction to generate acceptable solutions. This abstraction is related to the counter definition in an EFSM. In our previous work [5], we classified the assignment operations *op* to three types op^{vp}, op^{vv} and op^{vc} which denote a variable is assigned a value based on a parameter, variable(s), and a constant respectively. We further classify the operation op^{vv} to four subtypes to cover the counter situations: (1) op^{v+c}: it increments v by a constant value. (2) op^{v-c}: it decrements v by a constant value. (3) $op^{v\times c}$: it multiplies v by a constant value. (4) $op^{v/c}$: it divides v by a non-zero constant value.

Based on the classifications of counter situations, we can provide the following definitions:

Definition 1: A context variable v in an EFSM is a counter variable if there is a transition t with assignment $op \in \{ op^{v+c}, op^{v-c}, op^{v\times c}, op^{v/c}\}$ to v and v is referenced by a guard.

Definition 2: A transition t is affecting a counter variable v if it assigns to v using an $op \in \{op^{vc}, op^{v+c}, op^{v-c}, op^{v\times c}, op^{v/c}\}$, however, t is affected-by a counter if it has a guard that references v.

Definition 3: A transition t is an *initialiser* of a counter variable v if it assigns to v *a* constant using an $op \in \{op^{vc}\}$, however t is an *updater* of v if it assigns to v using an $op \in \{op^{v+c}, op^{v-c}, op^{v\times c}, op^{v/c}\}$.

Definition 4: Given a counter variable v, a TP: $t_1, t_2, .., t_n$ and three transitions t_i, t_j and t_k from this TP where $i < j < k$, the triple (t_i, t_j, m) forms a sequence that can satisfy t_k guard if t_i is an initialiser of v, t_j is an updater of v, t_k is an affected-by v, the path $t_i,..,t_k$ contains exactly m instances of transition t_j without other assignments to v and $m \geq 0$ is an integer that specifies the exact number of updater occurrences so that t_k guard is satisfied.

Based on the above definitions, if a given TP is required to cover a transition t_k which is affected-by a counter, the problem can be approached by having a method to automatically determine all the possible triples of (initialiser, updater, updater times) that can satisfy the guard of t_k.

A. Dependencies Representation

For each counter variable v, we determine its initialisers and updaters. Also, we record the type of operation of each updater and the guard type of each transition that is affected-by a counter variable. Table 1 and Table 2 show the integer representation of possible guard operators and operations. Based on this representation, we can construct two matrices: affecting a counter *aff*[] and affected by a counter *aff-by*[]. Each row in these matrices represents one transition and each column represents one counter variable. Each cell comprises a tuple with two fields: one to record the operation code in case of an *aff*[] matrix or the guard code in case of *aff-by*[] matrix. The other field of the tuple records the value of the constant that appears in the operation or the guard.

B. Finding the Required Sequence of Transitions

The proposed approach is applied through an algorithm which consists of three routines: **FindSequence, Validate** and **GuardCheck**. The first procedure, **FindSequence,** is the main one and it determines whether a target transition, t_j, has guard(s) referencing a counter variable(s). Also, it scans for all possible (initialiser, updater) pairs that initially have the potential to satisfy the given target guard(s) over a counter variable(s). The second function, **Validate**, is called from the main procedure to validate each given triple (initialiser, updater, updater times) by: (1) checking whether calling only the initialiser can satisfy the target transition guard and (2) setting a given triple as either invalid or valid and thus setting the number of times the updater must be called. The last function, **GuardCheck**, is used by function **Validate** to check if a given guard is satisfied. Fig. 1 shows a high level description of the algorithm that finds a sequence of transitions to bypass the counter problem.

Naturally, there can be more than one possible triple (initialiser, updater, updater times) that may satisfy t_j guard. For such a case, a set of triples are linked by *OR*. If t_j guard references more than one counter variables, then triples that belong to different counter variables are linked by *AND*.

Once the final set of triples is ready, it is fed to the TP fitness metric [5] to check whether the transition to cover (t_j) exists, furthermore, the triple (initialiser, updater, updater times) is present in this TP. This is accomplished by checking that the initialiser comes first, then the updater occurs certain number of times and finally the target t_j. The path from the initialiser to the first occurrence of the updater must be definition clear for the related counter variable. The occurrences of the updater need not be in successive transitions in the path and it is sufficient that the sub-path between each pair of occurrences of the updater is definition clear for the counter variable. Finally, the path from the last occurrence of the updater to the target t_j must also be definition clear for the same counter variable. If a given generated TP does not meet the criterion, then a large penalty value (10000) is assigned i.e. infeasible TP.

In order to verify whether a generated TP is feasible, a method is required to trigger this TP. If a TP requires a set of test data, then a suitable set of test data should be provided. In this study, we use the test data generator technique proposed in [8] to trigger the generated TPs.

III. EXPERIMENT AND APPROACH VALIDATION

In this paper, we use two EFSM case studies to validate the proposed approach. The first EFSM is the Inres initiator and the second is an ATM system. The specification details of these EFSMs are given in [7].

Procedure find sequences of transitions

```
1.  input: tj, affecting matrix aff[], affected-by matrix affby[]
2.  output: a list LT of triples (initialiser, updater, updater_times)
3.  goal: determine a sequence of transitions to satisfy tj guard
4.  initialize variable: empty(LT)
5.  for vi = 1 to number_of_counter_varaibles
6.    if gurad of tj references a counter_var_vi then
7.      build a list LI of transitions that initialise vi
8.      build a list LU of transitions that update vi
9.      for every initialiser from LI
10.       for every updater from LU
11.         if validate (initialiser, updater, updater_times) then
12.           insert in LT the triple (initialiser, updater, updaterTimes);
13.         endIf
14.       endFor
15.     endFor
16.   endIf
17.endFor
```

Function guardCheck

```
a1. input: guard_code, guard_const, counter_var
a2. output: Boolean result
a3. goal: check whether a guard is satisfied
a4. Initialize variable:  result = false;
a5. case guardCode of
a6.   0 : result = true; // no guard
a7.   1 : if counter_var = = guard_const then result = true;
a8.   2 : if counter_var > guard_const then result = true;
a9.   3 : if counter_var < guard_const then result = true;
a10.  4 : if counter_var ≥ guard_const then result = true;
a11.  5 : if counter_var ≤ guard_const then result = true;
a12.  6 : if counter_var ≠ guard_const then result = true;
a13.endCase
```

Function validate

```
b1.  input: tj, vi, aff[], affby[], a triple (initialiser, updater, updater_times)
b2.  output: Boolean result
b3.  goal: determine how many times an updater should be called in order to
         satisfy the guard of tj over the counter variable vi
b4.  initialize variable: result = false;        // currently set the triple to be invalid
b5.  counter_var_vi = initialiser_const;       // apply the initialiser operation
b6.  if guardCheck(tj_guard_code, counter_var_vi, tj_guard_const) then
b7.    updater_times := 0;     // calling the initialiser alone can satisfy tj_guard
b8.    result = true;             // and so the triple is valid; exit the routine
b9.    exit;
b10.endIf
b11. repeat
b12.   first_branch_distance_of_tj_guard = | counter_var_vi - tj_guard_const |;
b13.   if guardCheck(updater_guard_code, counter_var_vi, updater_guard_const) then
         // check if the updater guard is true and so the updater can be called
b14.     case updater_operation_code of
b15.       2 : counter_var_vi = counter_var_vi + updater_operation_const;
b16.       3 : counter_var_vi = counter_var_vi - updater_operation_const;
b17.       4 : counter_var_vi = counter_var_vi * updater_operation_const;
b18.       5 : counter_var_vi = counter_var_vi / updater_operation_const;
b19.     endCase
b20.   updater_times = updater_times + 1;
b21.   else  result = false;    // cannot call the updater and thus the triple is invalid
b22.         break;
b23.   endIf
b24.   if guardCheck(tj_guard_code, counter_var_vi, tj_guard_const) then
b25.     result = true;   // triple is valid; no more calls to the updater is required
b26.     break;            // break and exit
b27.   else                  // so far tj guard is not satisfied, check if a loop can reoccure
b28.     next_branch_distance_of_tj_guard = | counter_var_vi - tj_guard_const |;
b29.   endIf
b30. until (next_branch_distance_of_tj_guard ≥ first_branch_distance_of_tj_guard)
       // the value that can satisfy tj guard has been surpassed and the triple is invalid
```

Figure 1. High level description of the algorithm that find a sequence(s) of transitions to cover a target transition with the counter problem

Since we study the counter problem, a TP length should be adequate to allow a certain transition, an updater, to occur a sufficient number of times. In this study, we consider TPs with length $n = 10$ to be long enough to allow generating a set of FTPs from both EFSMs. However, this TP length is used for an illustration purpose and longer TP length can be used. Also, we compare this approach with the previously proposed TP fitness metric [5] to understand how the proposed approach enhances the previous approach.

A GA search that implemented the previously defined TP fitness metric together with the proposed approach was applied to the two EFSM case studies to generate two sets of TPs (16 TPs for Inres initiator and 30 TPs for the ATM) that provide the transition coverage test suites (see Tables 3 and 4). During TP generation, each path was first checked for the existence of a particular transition that this TP is intended to cover, then if this TP included a guard that referenced a counter variable, then the test criterion (transition coverage) for this path was modified to require extra transitions (triples) and the TP was rechecked against these additional requirements. Any TP that violates these constraints was given a large fitness value (10000).

For the purpose of comparison, two similar alternative sets of TPs were also generated by using only the guide of the previously defined TP fitness metric. Since the complete set of results cannot fit in this paper, Table 5 reports only the alternative generated TPs that reference counter variables.

The GA searches (FTPs and path test data generation) were implemented using the publicly available Genetic and Evolutionary Algorithm Toolbox [9]. A detailed description of each of the tool parameters used is beyond the scope of this paper. However, these parameters are fully explained at the tool website [9] and we record the values used here for the purpose of experiment replication.

An integer valued encoding was used. The population size was 100 individuals where each individual consisted of 10 variables. The selection method was linear-ranking with a selective pressure set to 1.8. Discrete recombination was used whereas mutate integer mutation was applied. The range of values allowed was [0..1000] for path test data generation, [1..28] for Inres initiator FTPs generation and [1..60] for the ATM FTPs generation. Searches were allowed 1000 generations before being terminated. Finally, for path test data generation, we repeated the search 10 times for each subject TP.

A. *Experimental Results*

Table 3 shows that the entire 16 subject TPs for the Inres initiator were feasible. The TP fitness metric values associated with the TPs that did not suffer a counter problem were generally low. However, the TPs that include transitions with guards that reference a counter variable were associated with higher TP fitness values. This applies to TPs I_4, I_9 and I_{11}, however these paths are all feasible.

From Table 5, the alternative Inres TPs which reference a counter variable were associated with lower TP fitness

TABLE 3. SUBJECT TPS FOR THE INRES INITIATOR

ID	Subject TPs	Params	Fitness	Avg.Gen	Taken
I_0	$\mathbf{t_0},t_{12},t_{12},t_1,t_{13},t_{12},t_1,t_3,t_{13},t_{12}$	0	0	1	Yes
I_1	$t_0,t_{12},t_{12},\mathbf{t_1},t_{13},t_{12},t_1,t_{13},t_1,t_{13}$	0	0	1	Yes
I_2	$t_0,t_1,\mathbf{t_2},t_{14},t_{12},t_1,t_2,t_{14},t_1,t_3$	0	0	1	Yes
I_3	$t_0,t_{12},t_{12},t_1,\mathbf{t_3},t_2,t_{14},t_{12},t_1,t_{13}$	0	0	1	Yes
I_4	$t_0,t_{12},t_{12},t_{12},t_1,t_3,t_3,t_3,t_3,\mathbf{t_4}$	0	820	1	Yes
I_5	$t_0,t_{12},t_{12},t_{12},t_{12},t_{12},t_1,t_2,\mathbf{t_5},t_{10}$	0	0	1	Yes
I_6	$t_0,t_{12},t_1,t_2,t_5,t_7,t_5,\mathbf{t_6},t_5,t_{10}$	2	48	1	Yes
I_7	$t_0,t_1,t_{13},t_1,t_2,t_5,\mathbf{t_7},t_5,t_{15},t_{12}$	1	24	25	Yes
I_8	$t_0,t_1,t_3,t_2,t_{14},t_{12},t_1,t_2,t_5,\mathbf{t_8}$	1	6	9	Yes
I_9	$t_0,t_1,t_3,t_2,t_5,t_8,t_8,t_8,t_8,\mathbf{t_9}$	5	850	1	Yes
I_{10}	$t_0,t_{12},t_{12},t_1,t_{13},t_1,t_2,t_5,\mathbf{t_{10}},t_{15}$	0	0	1	Yes
I_{11}	$t_0,t_1,t_2,t_5,t_8,t_8,t_8,t_8,\mathbf{t_{11}},t_1$	4	844	1	Yes
I_{12}	$t_0,t_1,t_{13},t_1,t_3,t_{13},\mathbf{t_{12}},t_{12},t_1,t_{13}$	0	0	1	Yes
I_{13}	$t_0,t_1,t_2,t_{14},t_1,\mathbf{t_{13}},t_1,t_2,t_{14},t_{12}$	0	0	1	Yes
I_{14}	$t_0,t_{12},t_1,t_{13},t_1,t_2,\mathbf{t_{14}},t_{12},t_1,t_3$	0	0	1	Yes
I_{15}	$t_0,t_{12},t_{12},t_{12},t_{12},t_1,t_2,t_5,t_{10},\mathbf{t_{15}}$	0	0	1	Yes

TABLE 4. SUBJECT TPS FOR THE ATM

ID	Subject TPs	Params	Fitness	Avg.Gen	Taken
A_1	$\mathbf{t_1},t_4,t_6,t_{25},t_{26},t_{25},t_{26},t_8,t_{10},t_{25}$	2	36	31	Yes
A_2	$t_1,t_4,t_5,t_8,t_{10},t_7,t_9,t_{23},t_1,\mathbf{t_2}$	4	54	85	Yes
A_3	$t_1,t_2,t_4,t_6,t_{23},t_1,t_2,t_2,t_2,\mathbf{t_3}$	8	802	141	Yes
A_4	$t_1,\mathbf{t_4},t_5,t_{25},t_{26},t_{25},t_{26},t_{25},t_{26},t_7$	2	36	26	Yes
A_5	$t_1,t_4,\mathbf{t_5},t_8,t_{10},t_{25},t_{26},t_7,t_9,t_7$	2	36	23	Yes
A_6	$t_1,t_4,\mathbf{t_6},t_{25},t_{26},t_{25},t_{26},t_{25},t_{26},t_8$	2	36	26	Yes
A_7	$t_1,t_4,t_5,t_{25},t_{26},t_8,t_{10},t_{25},t_{26},\mathbf{t_7}$	2	36	23	Yes
A_8	$t_1,t_4,t_6,t_7,t_9,\mathbf{t_8},t_{10},t_8,t_{10},t_8$	2	36	34	Yes
A_9	$t_1,t_4,t_5,t_7,\mathbf{t_9},t_{25},t_{26},t_{25},t_{26},t_7$	2	36	25	Yes
A_{10}	$t_1,t_4,t_5,t_{25},t_{26},t_7,t_9,t_8,\mathbf{t_{10}},t_8$	2	36	27	Yes
A_{11}	$t_1,t_4,t_5,t_7,t_9,t_{25},t_{26},t_7,\mathbf{t_{11}},t_{16}$	4	60	101	Yes
A_{12}	$t_1,t_4,t_6,t_8,t_{10},t_7,\mathbf{t_{12}},t_{15},t_9,t_8$	4	60	128	Yes
A_{13}	$t_1,t_4,t_6,t_7,t_9,t_7,\mathbf{t_{13}},t_{16},t_9,t_8$	4	56	60	Yes
A_{14}	$t_1,t_4,t_6,t_7,\mathbf{t_{14}},t_{15},t_9,t_8,t_{10},t_{25}$	4	76	82	Yes
A_{15}	$t_1,t_4,t_6,t_{25},t_{26},t_7,t_{13},\mathbf{t_{15}},t_9,t_{25}$	4	56	64	Yes
A_{16}	$t_1,t_4,t_6,t_7,t_{13},\mathbf{t_{16}},t_9,t_{25},t_{26},t_{25}$	4	56	81	Yes
A_{17}	$t_1,t_4,t_6,t_8,\mathbf{t_{17}},t_{22},t_{10},t_{25},t_{26},t_7$	4	72	75	Yes
A_{18}	$t_1,t_4,t_5,t_8,\mathbf{t_{18}},t_{21},t_{10},t_{25},t_{26},t_8$	4	56	60	Yes
A_{19}	$t_1,t_4,t_5,t_{25},t_{26},t_8,\mathbf{t_{19}},t_{22},t_{10},t_7$	4	60	382	Yes
A_{20}	$t_1,t_4,t_6,t_7,t_9,t_8,\mathbf{t_{20}},t_{22},t_{10},t_{25}$	4	60	222	Yes
A_{21}	$t_1,t_4,t_5,t_8,t_{10},t_8,t_{18},\mathbf{t_{21}},t_{10},t_7$	4	56	57	Yes
A_{22}	$t_1,t_4,t_5,t_{25},t_{26},t_{25},t_{26},t_8,t_{18},\mathbf{t_{22}}$	4	56	64	Yes
A_{23}	$t_1,t_4,t_5,t_{25},t_{26},t_8,t_{10},t_7,t_9,\mathbf{t_{23}}$	3	48	60	Yes
A_{24}	$t_1,t_4,\mathbf{t_{24}},t_1,t_4,t_6,t_{25},t_{26},t_8,t_{10}$	4	72	164	Yes
A_{25}	$t_1,t_4,t_5,t_7,t_9,\mathbf{t_{25}},t_{26},t_8,t_{10},t_{25}$	2	36	41	Yes
A_{26}	$t_1,t_4,t_6,t_{25},t_{26},t_7,t_9,t_{25},\mathbf{t_{26}},t_7$	2	36	30	Yes
A_{27}	$t_1,t_4,t_6,t_8,t_{10},t_{25},\mathbf{t_{27}},t_{30},t_{26},t_8$	6	98	300	Yes
A_{28}	$t_1,t_4,t_5,t_{25},\mathbf{t_{28}},t_{30},t_{26},t_7,t_9,t_{25}$	6	98	175	Yes
A_{29}	$t_1,t_4,t_6,t_{25},t_{26},t_{25},t_{26},t_{25},t_{27},\mathbf{t_{29}}$	6	98	464	Yes
A_{30}	$t_1,t_4,t_5,t_{25},t_{27},\mathbf{t_{30}},t_{26},t_{25},t_{26},t_7$	6	98	317	Yes

metric values than that observed in Table 3, however, these were not successfully triggered. This relates to paths Ii_4, Ii_9 and Ii_{11} which are infeasible due to insufficient occurrence of a specific transition (the updater).

From Table 4, we also find that the subject TPs derived from the ATM were all feasible. In addition, the path A_3 is associated with the highest TP fitness metric value due to this path referencing a counter variable. Compared to the alternative ATM set of TPs which reference a counter variable (see Table 5), the path Aa_3 is associated with lower TP fitness metric but this path is infeasible due to the insufficient occurrences of the updater.

The results show that for these EFSMs the proposed approach was successful in generating FTPs to provide

TABLE 5. ALTERNATIVE SUBJECT TPS

ID	EFSM	Subject TPs	Params	Fitness	Avg.Gen	Taken
Aa_3	ATM	$t_1,t_2,\mathbf{t_3},t_1,t_4,t_5,t_7,t_9,t_7,t_9$	4	208	1000	No
Ii_4	Inres	$t_0,t_1,t_{13},t_1,t_3,\mathbf{t_4},t_{12},t_1,t_{13},t_{12}$	0	136	1000	No
Ii_9	Inres	$t_0,t_1,t_2,t_5,t_{10},\mathbf{t_9},t_1,t_{13},t_{12},t_1$	1	142	1000	No
Ii_{11}	Inres	$t_0,t_1,t_{13},t_{12},t_{12},t_1,t_2,t_5,t_{10},\mathbf{t_{11}}$	0	136	1000	No

100% transition coverage even though the EFSMs suffered from the counter problem. These results clearly show that the proposed FTP generation approach has enhanced the TP fitness metric to successfully bypass the counter problem.

IV. CONCLUSION

In this paper, we present a novel approach to generate FTPs from EFSMs that suffer from the counter problem. The proposed approach extends our previous FTPs generation approach. The proposed approach was utilized to guide a GA search to find two sets of TPs that provide the transition coverage test suites for two EFSM case studies which suffer from the counter problem. The experimental results show that the proposed approach effectively guided the GA search towards TPs that were feasible and the counter problem was successfully bypassed. Future research will investigate other cases of the counter problem that might exist, though in practice, the counter problem is likely to appear in the ways described in this study.

REFERENCES

[1] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification & Reliability*, vol. 14, pp. 105-156, 2004

[2] A. Petrenko, S. Boroday, and R. Groz, "Confirming configurations in EFSM testing," *Software Engineering, IEEE Transactions on*, vol. 30, pp. 29-42, 2004.

[3] S. T. Chanson and J. Zhu, "A unified approach to protocol test sequence generation," *Proceedings of 12th Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future. IEEE*, vol. 1, 1993, pp. 106-114.

[4] A. Y. Duale and M. U. Uyar, "A method enabling feasible conformance test sequence generation for EFSM models," *Computers, IEEE Transactions on*, vol. 53, pp. 614-627, 2004.

[5] A. S. Kalaji, R. M. Hierons, and S. Swift, "Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)," in *Software Testing, Verification, and Validation (ICST), 2nd IEEE International Conference on*, IEEE, 2009, pp. 230-239.

[6] M. Harman, "Open Problems in Testability Transformation," in *Software Testing Verification and Validation Workshop. IEEE International Conference on*, IEEE, 2008, pp. 196-209.

[7] A. S. Kalaji, R. M. Hierons, and S. Swift, "A search-based technique for testing from extended finite state machine model," Brunel University, Technical report, 2009, pp. 1-36. Available from: http://bura.brunel.ac.uk/handle/2438/3624.

[8] A. S. Kalaji, R. M. Hierons, and S. Swift, "A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine (EFSM)," in *Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART)*, IEEE, 2009, pp. 131-132.

[9] H. Pohlheim, "GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab," 1994-2010. http://www.geatbx.com.