Multi-objective construction of an entire adequate test suite for an EFSM

Nesa Asoudeh
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
nasoudeh@sce.carleton.ca

Yvan Labiche
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
labiche@sce.carleton.ca

***Abstract*—In this paper we propose a method and a tool to generate test suites from extended finite state machines, accounting for multiple (potentially conflicting) objectives. We aim at maximizing coverage and feasibility of a test suite while minimizing similarity between its test cases and minimizing overall cost. Therefore, we define a multi-objective genetic algorithm that searches for optimal test suites based on four objective functions. In doing so, we create an entire test suite at once as opposed to test cases one at a time. Our approach is evaluated on two different case studies, showing interesting initial results.**

***Keywords*—**state-based testing; EFSM; genetic algorithm; multi-objective optimization; case studies

I. INTRODUCTION

Extended Finite State Machines (EFSMs) are widely used in software modeling and a great volume of research exists in the area of state-based testing from EFSMs. One of the main difficulties in test generation from an EFSM is that not all the paths in an EFSM are feasible [1], because of guard conditions and actions in the EFSM. Another challenge is decreasing cost and increasing effectiveness of generated test suites to make them scalable to large industrial applications. The latter can be achieved for instance through increasing test case diversity [2].

Search-Based Software Engineering (SBSE) has emerged in the field of software engineering since the nature of software engineering problems lend themselves well to meta-heuristic search techniques, especially when tradeoffs between different constraints need to be found, since they can provide solutions in cases where optimal solutions are either theoretically impossible or practically infeasible [3]. SBSE has proved to be very successful and there has been a significant increase in interest in this field in the recent years [4]. In search-based software testing (SBST) a meta-heuristic search automates or partially automates a testing task [5].

In this paper, we propose an SBST technique for test suite generation from EFSMs. We use a multi-objective genetic algorithm (GA) to search for an adequate test suite (so far for the all-transitions criterion) that is most likely feasible, has low cost, and has low similarity between its test paths. To the best of our knowledge, this is the first attempt to use those four objectives together in the context of state-based testing (SBT). Another important contribution is that we attempt to build an entire test suite at once as opposed to constructing test cases one at a time until adequacy is reached, as has been done in the past during SBT. We argue that the latter is a kind of greedy search for an adequate test suite and therefore can only lead to a sub-optimal solution to the problem of adequate test suite construction.

The rest of the paper is structured as follows. Section II discusses related work. Section III describes our multi-objective genetic algorithm, which we experiment with as reported in section IV. We discuss threats to the validity of our results in section V. Conclusions are drawn in section VI.

II. RELATED WORK

Testing strategies put into place when testing from a state machine, or a set of communicating state machines, heavily depend on the kind of behaviour specification the state model contains. When the state machine does not have actions (or activities) on transitions or states, or guard conditions, then approaches exist to automatically generate feasible test cases, such as the W-method [6]: any traversal of the graph representing the state machine is feasible. A set of communicating state machines can under some conditions (to avoid a state space explosion) be transformed into a larger EFSM from which the abovementioned techniques can be used [7]. Alternatively, techniques specific to communicating state machines exist (e.g., [8]). Other techniques involve symbolic execution (e.g., [9], [10]) or a meta-heuristic search (e.g., [11]). Alternatively, one can consider testing techniques for labeled transition systems (e.g., [12]).

When the state machine (or a set of communicating state machines) has actions (or activities) and/or guards that are all linear (i.e., they can be written in the form $c_1x_1+ c_2x_2+\ldots+ c_nx_n = b$, where c_i and b are constants) [1], automated test case construction is also feasible: e.g., [1], [13], [14], [15], [16]. In all those cases, the testing technique is typically offline since it is possible to statically analyze the model and create feasible test cases prior to executing them. There are exceptions, such as UPPAAL-TRON [13], which is a technique (and a tool) for online testing a real-time embedded software from a timed automaton (or network of timed automata) specification.

In previously mentioned approaches, generated test cases typically have the form prefix-target-suffix [17], where *target* is a test objective to achieve, such as a transition to reach, *prefix* is a path in the state machine that allows the test to reach the target, and *suffix* is a sequence that allows for instance to identify the state that was reached after triggering the target. Test cases are therefore typically created one at a time, each test case achieving one test objective. Instead, we try to generate an entire adequate test suite in one optimization step.

When actions (or activities) and guards are specified with a more complex language, offline testing is typically not possible since it is very difficult (or even impossible) to statically analyze both the state-based behavior and the pieces of code, which can be as complex as any piece of Java/C/C++ code, to create feasible test cases: e.g., Conformiq [10] uses a Java-like language for specifying actions; Instead, online testing is necessary to simulate state changes caused by those pieces of code to identify the resulting state and therefore identify what can be the next event to send to the implementation/simulation of the state–based behavior.

Others before us have applied SBST to SBT, most notably [2], [14]. Hemmati et al. proposed a similarity based test case selection/minimization technique to maximize test case diversity as diversity is believed, and has been shown to relate to effectiveness at finding faults [2]. In a series of case studies they used different similarity measures as objective function and considered different search based algorithms: greedy, clustering-based, hill climbing, Evolutionary Algorithm (EA), and genetic algorithm (GA). By comparing the effectiveness at detecting faults and the cost of selected test cases, they conclude that EA, which is a simplified version of GAs, and the Gower-Legendre (set based) similarity measure is the best combination of optimization technique and similarity measure. Kalaji et al. [14] generate (likely) feasible test paths one at a time from an EFSM to produce an all-transitions adequate test suite, and then generate input sequences that trigger those test paths. Two different GAs are used for these two steps. The objective function of the first GA uses a data flow analysis of the state machine. An affecting/affected-by transition pair is a pair of transitions such that the first transition defines a variable, the second transition uses that variable and there is a definition-clear path in the state machine between the two. Each identified pair in the EFSM is assigned a penalty based on the type of assignment operation and comparison statement that appear in the affecting and affected-by transitions (see [14] for more details). The objective function of the second GA is an adaptation of similar work for finding test data for white box testing [18], and is based on the notions of branch distance and transition approach level. The branch distance represents how close an input value is to satisfying a specific predicate (in their case a guard condition). The transition approach level measures how close a test path is to executing a specific statement (in their case a transition). Their technique is empirically validated on several case studies.

SBST has been mostly used in structural testing [5], initially considering test data generation a single-objective optimization problem, focussing on one test objective (e.g., a branch in a control flow graph) at a time. Different avenues have been researched to improve such test data generation. On the one hand, Harman et al [19] proposed the first formulation of test data generation as a multi-objective problem. On the other hand, Michael et al [20] extended a single-objective search to build an adequate test suite achieving many test objectives at once, instead of searching for one solution for each test objective separately, and iterating over objectives: The authors do build an adequate test suite one test case at a time; however, as opposed to other approaches, the GA-supported search for a new test case is guided by what has already been covered by already created test cases. A third avenue is to create a whole adequate test suite in one general multi-objective search, instead of creating a test case for each test objective [21]. All these approaches consider structural testing while we consider state-based testing. The application of SBST to functional testing and specifically state-based testing has been very limited [14] and has not yet considered more than one objective, to the best of our knowledge.

In summary, our approach differs from previous contributions in the following aspects. (1) With respect to the construction of an entire test suite, we conduct functional testing rather than structural testing, and we use four objective functions together. (2) In the domain of SBT, we create an adequate test suite at once rather than test cases one at a time to achieve sub-objectives. We believe that creating test cases one at a time is a sub-optimal strategy/optimization, and that our construction of a complete test suite is a more global optimization. (3) We use a multi-objective algorithm to simultaneously account for test case diversity within the test suite, feasibility of individual test cases, cost and coverage of the entire test suite. We note that addressing multiple test objectives is one of the open problems in search based software testing [5]. In other words, entire test suite construction is not novel in structural testing [21] but, as far as we know, *entire test suite construction for functional, state-based testing* is a novel contribution of this paper; multi-objective test case construction is not novel in structural testing but, as far as we know, *multi-objective test suite construction for functional, state-based testing* is a novel contribution of this paper.

III. PROPOSED APPROACH

This section presents our solution to the problem of generating a test suite for an EFSM. We observe that when a test practitioner (manually) generates test cases from an EFSM she typically has multiple goals in mind. Avoiding infeasible paths caused by conflicts between actions and guards is one of those goals. She will also want to achieve some level of coverage of the EFSM test model since (1) this is a way to trigger specific behaviours (e.g., transitions) and (2) using a selection criterion gives a stopping condition for the construction of test cases (one stops creating new test cases when adequacy is achieved). She would also trigger behaviours as much different as possible in different test paths, thus introducing variability in her test paths (suite), in an attempt to identify as many faults as possible, i.e., to increase effectiveness. For instance, she could try to avoid using the same sub-path several times in different test cases. She will also care for the overall cost of the testing activity. She might

also be interested in other goals, but the four abovementioned are likely the most important ones.

Therefore, we argue that test generation from an EFSM is a multi-objective optimization problem, with a set of objectives including coverage of the EFSM test model, variability in generated test paths, feasibility of the test paths, and overall cost of the generated test suite. We also argue that these objectives are likely competing with one another. For example, our intuition is that increasing coverage or dissimilarity between test cases increases cost; ensuring a path is feasible may require that we create long paths, which increases cost.

We first provide some background on multi-objective optimization with GAs (section III.A). We then describe the main components of our GA: encoding of the problem solutions, i.e., what a chromosome and its genes represent (section III.B), mutation and crossover operators (sections III.C and III.D), objective functions (section III.E). Section III.F discusses some GA parameters, and the construction of the initial population. To perform our experiments, we tailored the multi-objective GA framework included in the MATLAB Global Optimization Toolbox [22].

A. Multi-objective Genetic Algorithm

Our optimization is multi-objective [23] with four different objectives, whereby a solution is characterized by a vector $\vec{u} = (u_1,...,u_k)$ of objective function values (in our case, k=4). A vector (solution) dominates another vector $\vec{v} = (v_1,...,v_k)$ (denoted by $\vec{u} \prec \vec{v}$) if and only if *u* is partially less than *v*: $\forall i \in \{1,...,k\}, u_i \leq v_i \wedge \exists i \in \{1,...,k\} : u_i < v_i$.

The objectives being optimized often conflict, placing a partial ordering on the search space. GAs are well suited to the task of finding a solution in such a situation [24] as they rely on a population of solutions: Individuals in the population represent solutions that are close to an optimum and provide different tradeoffs among the various objectives. The set of tradeoffs, which are solutions whose corresponding vectors are non-dominated by any other solution vector, is referred to as the Pareto front. The final decision with respect to which solution to select from the Pareto front is left with a decision maker, rather than the optimization algorithm.

B. Chromosomes and genes

A chromosome is a solution to the optimization problem [25], that is, in our context an entire test suite, which is a set of test cases. The genes compose a chromosome (test suite), and the length of a chromosome is its number of genes. A gene is therefore a test case, that is, a sequence of transitions of the state model (a.k.a. test path). One of the objectives of test suite construction is to achieve a certain level of coverage according to a selection criterion. Since different adequate (i.e., satisfying a given criterion) test suites, with varying number of test cases, usually exist for a given state model, our GA has chromosomes of variable length, i.e., variable number of genes (i.e., variable number of test cases).

Using a variable length chromosome can lead to a phenomenon known as bloat [26], whereby chromosomes get unreasonably bigger and bigger because small negligible improvements in an objective value are obtained with larger solutions. To control for bloat we have put a limit on the total number of transitions in a test suite, similarly to others [21]. Following a trial and error procedure: we ran the GA a couple of times and observed the number of transitions in adequate test suites. We specified a limit of twice the number of transitions in the biggest adequate test suite. Also, recall that one of our objectives is minimizing the cost (i.e. size) of a test suite: all other things being equal, a chromosome with a larger size will be penalized.

To encode a gene, i.e., a test path, we reuse a previously published solution as it facilitates the random construction of valid traversals of the graph representing the EFSM [14]. A test path is a valid traversal of the EFSM in the sense that this is a path in the graph representing the EFSM. This traversal may however not be feasible, i.e., we may not be able to find inputs to execute it. This is the reason why we use an objective function to increase chances of feasibility (see below). Our encoding of a chromosome is then a set of gene encodings.

C. Mutation Operator

Since in our context a chromosome has many characteristics (e.g., number of genes, length of each gene/test path, specifics of each gene/test path), a chromosome can be mutated in many different ways. To identify possible mutation operators, we systematically considered every single characteristic of a chromosome and investigated ways to mutate them.

We have defined seven different mutation operators, which are all applied with equal probability:

(1) Adding a gene to a chromosome, i.e., a randomly generated gene (i.e., test path) is added to the chromosome (test suite). Random generation of a gene (test path) proceeds as discussed by others [14] as this ensures we obtain valid traversals of the graph representing the EFSM, starting from the initial state. During this random construction, we randomly select the length of the test path to be constructed, between one and twice the length of the shortest path that covers all the states in the EFSM. If such a path does not exist, i.e., it is not possible to tour all the states in one path (i.e., the graph is disconnected), the length of the shortest path that tours the maximum number of states is considered. This is arguably a heuristic and it may happen that no feasible path (i.e., one can find inputs to execute it), even of that length can indeed feasibly reach some state or transition. However, we believe this leads to long enough newly randomly generated paths that other mutation or crossover operators can extend. We add this new gene (test path) to the chromosome (test suite) being mutated. In case the randomly generated gene is a duplicate of a gene already in the chromosome, we repeatedly try to generate a new one with the procedure above until the new one is different from an existing one.

(2) Removing a gene from a chromosome, i.e., a randomly selected gene (test path / test case) is removed from the chromosome (test suite);

(3) Replacing a gene, i.e., a randomly selected gene is replaced by a new randomly generated gene, following the procedure we discussed above in (1);

(4) Altering a test path (gene), which proceeds as follows. We randomly select a transition t_i in a randomly selected test path (gene) to be altered. This path can be represented in the form prefix.s_i.t_i.postfix, s_i being the start state of t_i. The operator is to randomly select a different sequence of transitions, possibly empty, from s_i, using the procedure discussed in (1) above. Once again, if the resulting test path is identical to an existing one, the operator is applied again until a new entire path is created.

(5) Appending a transition to a test path (gene), i.e., a randomly selected gene is mutated by randomly adding a transition to its end, ensuring the entire path is valid. If a path ends with a terminal state, from which there is no outgoing transition, this operator does not apply and another gene is selected for mutation.

(6) Changing a transition of a test path (gene), i.e., a randomly selected gene is mutated by randomly replacing one of its transitions with another one between the same two states, if another such transition exists, otherwise another gene is selected.

(7) Exchanging material between genes of the same chromosome, i.e., exchanging randomly selected sequences of transitions (sub-paths) between randomly selected test paths from the same test suite (chromosome), while ensuring that results are valid traversals of the graph representing the EFSM. To achieve this, we reuse a published procedure [14] since, once again, because we reuse their encoding, we obtain valid traversals. We note that when Kalaji et al. used this operator they were searching (by means of a GA) for a test case that reaches a specific transition. In their context, this operator is a crossover operator since the population manipulated by the GA is a population of test paths: the crossover exchanges material between chromosomes (test paths) of the population. In our case, the test paths involved in this operator are the genes of a chromosome. The operator therefore alters only one chromosome, and we classify it as a mutation operator in our context.

D. *Crossover operator*

Crossover creates two new chromosomes from two existing (parent) chromosomes by exchanging some genetic information, i.e., genes, from/between those parents. Once again, we asked ourselves what information can be exchanged between two (parent) chromosomes in our context. Our first crossover operator is a two-point crossover [25] in which the two points identify one gene in each parent instead of the more general case where they identify a sequence of genes. Given that we have variable length chromosome, we ensure the crossover points are valid: e.g., if a parent has 5 genes and the other parent has 10 genes, the crossover points identify one of the first five genes in both parents. We randomly select two parent chromosomes (test suites), p_1 and p_2, and one test path (gene) from each of the parents (at the same index in the sequence of genes of the parents). We remove the selected test path from p_1 and add it to p_2, and similarly remove the selected test path from p_2 and add it to p_1. As a chromosome represents a test suite, the order of its test paths does not matter. Therefore, we add the test path extracted from p_1 (resp. p_2) to the end of p_2 (resp. p_1).

Secondly, we can exchange genetic information at a lower level of details, specifically at the transition level between test paths. We select two parent chromosomes, select one gene (test path) in each of those parents, and exchange transition sequence information between those two genes. This second crossover operator works identically to the last mutation operator we discussed, except that the two genes (test paths) are selected from different chromosomes (test suites), whereas for mutation purposes the two genes belonged to the same chromosome.

When crossover happens, with a specific probability (see below), these two crossover operators have an equal probability of occurrence.

E. *Objective functions*

As previously mentioned, we are considering four objective functions: Feasibility, to be maximized; Similarity, to be minimized; Coverage, to be maximized; and Cost, to be minimized.

To determine the **feasibility** of a test path, we reuse previous work that relies on an analysis of data flow dependencies between the transitions of a test path [14]. Different types of data flow dependencies that might exist between two transitions of a test path are assigned penalty values based on their possible effect on feasibility. This information is used to obtain a feasibility measure for each gene (test path) in a chromosome (test suite) and then we obtain a feasibility measure for the chromosome as the sum of the feasibility values of its genes. Feasibility of a chromosome c_i, made of n genes, is then:

$$feasibility(c_i) = \sum_{j=1}^{n} feasibility(g_j);$$

Where feasibility(g_j) is the feasibility of gene g_j and is obtained by using the approach proposed by Kalaji and colleagues [14].

It is important to note that feasibility(g_i) is only an estimation of the feasibility to find input values such that the path defined by g_i can actually be executed. Kalaji et.al. [14] have shown that this measure is indeed a surrogate measure of feasibility: one has more chances of finding inputs to execute test paths that have low feasibility values. However, actually trying to find inputs such that path g_i executes is the only way to identify whether g_i is indeed feasible.

Our intuition, as well as others' [2], is that test paths should be as dissimilar as possible to increase fault detection. We are therefore interested in computing the **similarities** between pairs of test paths (genes) to obtain a similarity value for a test suite (chromosome). Different similarity measures can be used as objective function. One of the measures [2], which is not limited to identical length sequences (we have variable length

chromosomes), is the Levenshtein distance [27]. Although not the recommendation of Hemati et al. (recall section II) this is the best measure of similarity they studied [2] that supports variable length sequences. To change this distance measure into a similarity measure, similar to Hemmati et al. [2], we reward each match between two sequences by one point and penalize mismatches and gaps by simply ignoring them (i.e., assigning no point). More sophisticated measures (e.g. Needleman-Wunsch), which penalize mismatches and gaps, can be used as well. To obtain a similarity measure for a test suite (i.e., chromosome), we proceed similarly to Hemmati et al. [2]. The similarity measure for a test suite (chromosome) is the sum of similarity measures computed for each unordered pair of test paths (genes) in that chromosome. The following objective function needs to be minimized:

$$similarity(c_i) = \sum_{pairs\ of\ genes\ (g_j,g_k)} similarity(g_j, g_k).$$

We note that our measure of similarity is not the best one as per the experiments reported by Hemmati et al. (recall section II). We however selected it since they experimented with it, showing it performs well, and more importantly (i) it allows us to account for test paths of variables lengths and (ii) we believe a sequence-based measure like the Levenshtein distance is more appropriate in our context than set-based distances like Gower-Legendre since we measure test cases that are paths (sequence of states and transitions) in an EFSM. We plan to experiment with different measurements of test cases and test suite similarity in our future work.

Different **coverage** criteria can be used as objective function. Although more demanding, but also more effective (at finding faults) criteria exist [28], [29], we selected the all-transitions criterion. The objective function is to minimize the number of distinct transitions uncovered by a test suite (chromosome), i.e., to maximize coverage.

The last objective is reducing the **cost** of a test suite as much as possible. A usual surrogate measure of cost is the size of a test suite [30]. The underlying assumption is that test cost is proportional to the test suite size. The notion of cost in the context of testing is complex as one may want to consider time to market or computer time usage as part of the equation [31]. In our context, test set size can be simply measured by counting the number of test sequences in a test set (e.g., [32]). However, not all test sequences contain the same number of transitions triggered. We could go further and say that not all sequences take the same time to execute. But without going to that level of detail, we could measure cost/size as the number of transitions triggered in the test set, summing up the number of transitions triggered in each test sequence. This objective function is to be minimized.

F. Genetic algorithm parameters

There are a number of factors that affect the success of a GA. We selected a population size of 200, which conforms to what has been suggested in the literature [33]: i.e., a value in range [30, 80] (we selected 50) multiplied by the number of objective functions. Based on results from previous studies [34] we selected a crossover rate of 0.7 and a mutation rate of 0.01. The Pareto Fraction parameter controls elitism in a multi-objective GA since it limits the number of individuals in the Pareto set (elite or tradeoff members). Based on a previous study [35], which suggests to set the maximum size of the Pareto set such that the ratio of the Pareto set over the entire population is between ¼ and 4, we set the maximum size of the Pareto set to 35% of the entire population. With respect to the stopping criterion, we reuse the one of the toolbox we relied on, and stopped the GA if the observed average change in any objective function value over 10 generations was less than e^{-4}.

The initial population is generated randomly, i.e. we start with a set of 200 (our population size) randomly generated test suites. Each test suite has a random number of randomly generated valid traversals (of variable length) of the graph representing the EFSM. When generating a test suite, we first randomly select the number of transitions N this test suite will have, between one and the maximum number of transitions we defined earlier to control for bloat (section III.B). We then incrementally create test paths as random traversals of the state machine graph until the cumulative number of transitions in those test paths reaches N. For the purpose of creating test paths, we add a reset transition from each state to the start state. When creating a test path we then iteratively, randomly and uniformly select an outgoing transition (including the reset) from the current state. Admittedly, this can encourage the generation of short sequences, especially when states have a small number of outgoing transitions. Our experiment results show however this does not happen thanks to objectives which favour diversity.

IV. EXPERIMENTAL EVALUATION

In an initial attempt to evaluate our test suite construction technique, we used two case study EFSMs, modeling real world applications, to answer the following questions:

RQ 1. Are there any benefits in terms of cost for instance, in generating a whole test suite at once rather than test paths separately in an incremental manner?

RQ 2. Since we use a surrogate measure of feasibility during optimization, do we obtain actually feasible test suites? In other words, can we find inputs such that test suites in the Pareto front can actually execute?

RQ 3. Is our multi-objective GA successful, compared to random generation for instance, in addressing our optimization goals? In other words, is the search space so large or complex that a random search is not adequate and we have to resort to a GA?

RQ 4. How do test suites created by our GA compare to test suites created by existing approaches in terms of effectiveness at detecting faults?

The first EFSM models a simple cruise control system, which we have used in the past (e.g., [28], [35]) while the second EFSM is a simplified model of a class II transport protocol we obtained from others [14]. They are used to answer our research questions as summarized in TABLE I.

TABLE I. ANSWERING RESEARCH QUESTIOONS WITH CASE STUDIES

	RQ1	RQ2	RQ3	RQ4
Cruise Control	✓			✓

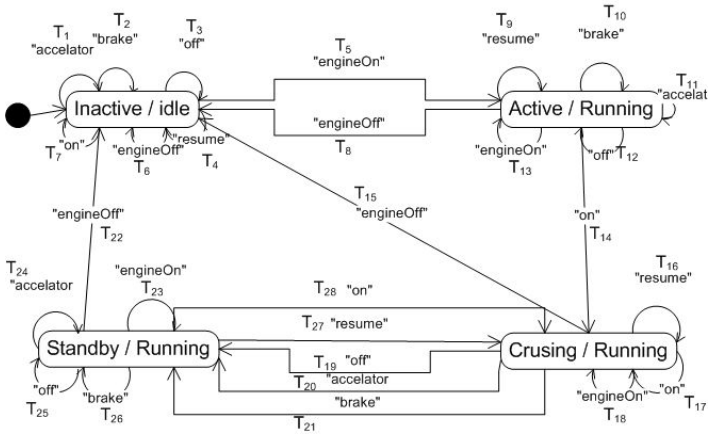

Fig. 1. EFSM for the cruise control system

Transport Protocol	✓	✓	✓	✓

Section IV.A introduces the two case studies while section IV.B discusses experimental design. Before answering research questions in sections IV.D to IV.G, we perform a qualitative analysis of some of the results we obtain with the Cruise Control case study: section IV.C.

A. Case study systems

We selected a simple model of a Cruise Control as a first case study because, although it does not have guards or actions (Fig. 1) and therefore any traversal of the state machine graph is a feasible test path (as a consequence we have really three objective functions), it allowed us to check the correctness of our approach and focus on the three other objective functions. With only three objective functions we can furthermore plot results and learn from those results in a qualitative analysis: e.g., we can visually observe the result of the competition between the objective functions. The state machine has four states and 28 transitions (Fig. 1).

The second EFSM is a simplified model of a Class II transport protocol [14], that models connection establishment, data transfer, end-to-end flow control and segmentation. The state machine has six states and 21 transitions (Fig. 2). The reader interested in more technical details about this model is referred to [14], where a complete description of the states, transitions, guards and actions is available. We selected this model because of two reasons: (i) There are guard conditions and actions in the model so not every valid traversal of the state graph is feasible; (ii) Despite guards and actions, we can automatically find feasible test paths in this state machine [14].

Although these two case studies represent two typical cases where a state machine is used to model behaviour, they are admittedly small. However, similar behaviors are usually modeled using state machines in UML-based development [36], [37], and in industrial case studies reported in the literature [38], [39]. Additionally, it is very uncommon in practice to model subsystems or entire systems using state machines, as this is far too complex in realistic cases. Last, the second case study is a representative sample of a series of case studies used by others [14].

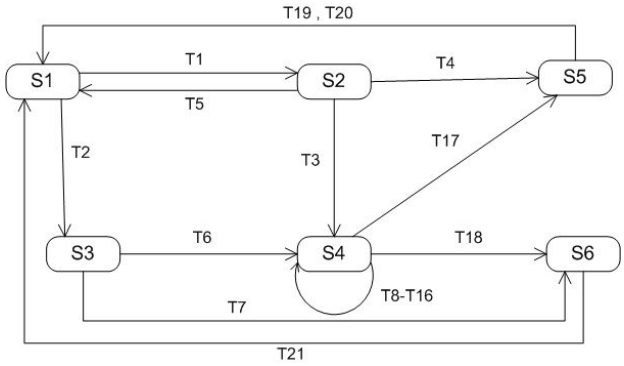

Fig. 2. EFSM of a class II transport protocol

B. Experimental design

1) Answering RQ 1

We simulated the construction of an all-transitions adequate test suite for Cruise Control and Transport Protocol by following a procedure similar to many existing approaches (recall section II) whereby we created a test case for each transition: each test case is made of a prefix followed by the target transition, the prefix being a shortest path to reach that transition. We intend to compare this test suite, which we refer to as T_{trad}, with adequate test suites produced by our GA in terms of cost, similarity and effectiveness at finding faults. Arguably, the test suite thus created may have redundant test cases, as pointed out by others (e.g., [14]): e.g., a test path targeting a transition may be a sub-path of a longer test path targeting another transition. To make the comparison more objective, we then removed the redundancy by iteratively removing any test path t_i that is a sub-path of another test path t_j in the test suite, until no further test path deletion is feasible. We obtained a test suite we refer to as T_{red}.

2) Answering RQ 2

As discussed earlier, every traversal of the state graph of the Cruise Control case study is feasible. So we only used the Transport Protocol case study to answer RQ 2. Using this case study, we investigated the feasibility of a sample of test suites in the Pareto front we obtain with our GA by manually trying to identify test inputs that would make their test paths to execute. Future work will investigate ways to automate this process, for instance by using a solution based on a genetic algorithm [14].

Specifically, at the end of a typical run of our GA, we ranked the test suites in the Pareto front in increasing order of (lack of) coverage: the first ones we obtained were adequate test suites. We then selected one sample test suite for each of the first six values of (lack of) coverage: 0, 1, 2, 3, 4 and 5 (these are the numbers of un-covered transitions by the test suites). For each of the six test suites, we tried to find input values such that the test cases they contain can actually execute.

3) Answering RQ3

We used the Transport Protocol case study only to answer this research question, because the model has guard and actions, and compare our test suite generation technique to a randomly generated test suite. To have an as unbiased as

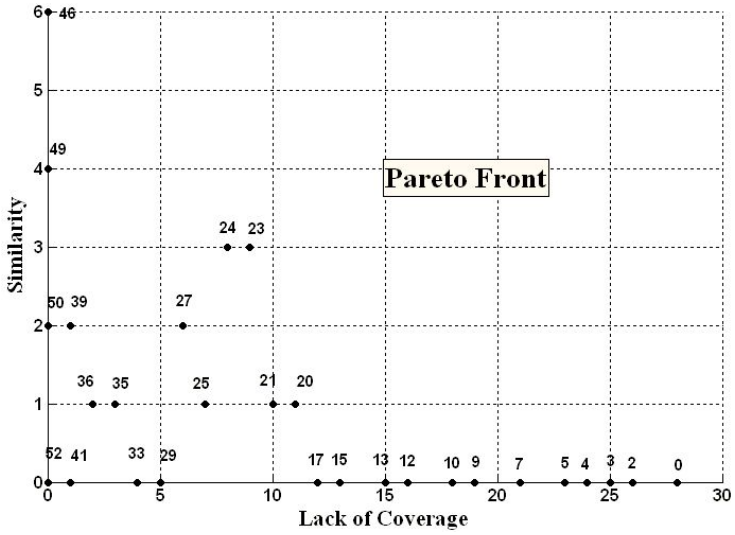

Fig. 3. Sample Pareto front contents for the Cruise Control: each plot is an element of the Pareto front showing lack of coverage (x-axis), similarity (y-axis), and cost (value next to plot).

possible comparison we considered the following when designing the random generation routine: (i) the random generation should investigate as many solutions in the search space as the number of solutions a typical GA run investigates; (ii) the random generation should investigate solutions that are comparable in terms of cost with solutions the GA investigates. Without (i) we would favour one technique over the other since one technique could easily perform better than the other simply because it considers more solutions. Since increasing coverage, diversity, and (likely) feasibility can be achieved by increasing cost, we would also favour one technique over the other without (ii).

To account for these two characteristics of the search, we monitored a typical execution of our GA and collected the following information: the number of new solutions being constructed, the cost (cumulative number of transitions) of each such new solution. With the latter we obtained the distribution of the number of test suites of specific cost values.

We then randomly generated as many solutions (test suites) as the number investigated by the GA and selected an overall cost for each randomly created test suite by randomly sampling the abovementioned distribution. Given an overall cost value, we generated test paths in the test suite by following the same procedure as the one we used to create an element of the initial population of our GA.

4) Answering RQ4

Mutation analysis is a well known method commonly used for the evaluation of testing strategies [40]. We used *MAJOR* [41], a mutation analysis tool for Java, to seed faults and perform the mutation analysis. We used all the mutation operators available with MAJOR and created 198 mutants for cruise control and 870 mutants for transport protocol. When identifying whether a mutant is killed or alive, we need an oracle for each test case. Our oracle is to check the value of state variables after each transition has fired in each test case.

We created other test suites using existing testing strategies, specifically: round trip path [42] because it has been shown to be a good alternative between the transition and transition pair selection criteria [29], a modified version of round trip path that attempts to increase diversity among test paths [28], a test suite manually generated, and the approach that creates one test path at a time to achieve transition coverage (recall RQ1). We compared the generated test suites with test suites created by our multi-objective GA in terms of effectiveness at detecting faults as well as our four objectives (i.e. cost, coverage, similarity and feasibility).

The manually generated test suite, referred to as T_{man}, was created by an engineer other than the authors of this paper: the engineer was not given any other instructions than to produce an adequate test suite; the result is an adequate test suite that minimizes at the same time overall cost and the number of test cases as well as their feasibility (in the case of Transport Protocol).

We note that in the two round trip paths test suites (the original one and the modified one) a couple of test cases were unfeasible because of guards and actions. The test cases systematically created by the generation algorithms were slightly modified (appending a transition to two paths) to ensure adequacy of the transition criterion.

5) Stochastic aspect of our GA

We also note that, since a GA is inherently a stochastic process, we would ideally need to run our GA a number of times on each case study and evaluate trends on a large number of resulting adequate test suites. We ran our GA a number of times on each case study and, although we do not report on trends, we observed results were similar over multiple runs. Due to space constraints, for each case study we therefore report on only one representative run and defer reporting on trends to a future publication.

C. Qualitative analysis of some results

Fig. 3 plots lack of coverage versus similarity for each test suite belonging to the Pareto front at the end of one (typical) run of our GA for the Cruise Control. This figure is a two-dimensional projection of a three-dimensional plot. Numerical values above each point represent the cost of the corresponding test suites (the third objective function). Test suites (points) right on the y-axis are all-transitions adequate.

Fig. 3 shows a test suite (point) with values (0, 0, 52) for coverage, similarity, and cost, respectively. This is an all-transitions adequate test suite with ideal (0) similarity value between its test cases. We refer to this test suite as TSga1. The other three adequate test suites in the Pareto front are referred to as TSga2, TSga3, and TSga4, in increasing order of their similarity value (2, 4, 6, respectively): e.g., TSga4 is (0, 6, 46). These cost values are higher than for other all-transitions or transition-tree adequate test suites we have created for this case study (section IV.G and [29]): the cost values we obtained, though not trying to obtain dissimilar test cases (which increases cost), where 25 for the all-transitions adequate test suites and 38 for the transition-tree adequate test suites.

Allowing coverage or similarity objectives to get further from their optimum value reduces cost. For example test suite TSga2, i.e., (0, 2, 50), has more similar test cases than TSga1 (0, 0, 52) at a slightly smaller cost; test suite (1, 0, 41) is not adequate (it misses one transition) but has a lower cost than TSga1.

We note that since cost and dissimilarity are objectives to be minimized and an empty test suite has ideal similarity, the Pareto front contains an empty test suite (Fig. 3): (28, 0, 0). Future work will look into avoiding such a solution.

When only using two objective functions, i.e., cost and coverage, we obtain the Pareto front of Fig. 4. What is noticeable is that the maximum cost is 39, whereas (Fig. 3) it is 52 when also accounting for similarity. This confirms one's intuition that improving coverage or improving dissimilarity between test cases increases cost. Studying to what extent these objective functions actually compete with one another will be part of our future work.

The last two observations may suggest that we may need to specify weights on our objective functions, to for instance promote coverage. This will be part of our future work.

Similar observations about competing objectives can be made for the Transport Protocol case study: see sample data in TABLE II. : e.g., decreasing coverage (recall we measure lack of coverage) decreases cost. Coverage level is gradually getting further from its optimal value while the total feasibility penalty of test suite is decreasing. This is due to the fact that not covering some transitions (especially the ones that are difficult to cover) increases the likelihood that test cases will be feasible. Also, decreasing coverage while maintaining a similar level of similarity decreases cost. We observed the same trade-offs for the Cruise Control.

D. Results for RQ 1

Recall that RQ 1 is to study the benefits and drawbacks of building an entire test suite at once rather than in a stepwise manner, one test case at a time, each test case achieving only one test objective (reaching one target transition), which we answer by using the Cruise Control as discussed in section IV.B.1).

Since, as mentioned previously, the Cruise Control EFSM has 28 transitions, T_{trad} is all-transitions adequate with 28 test cases [43]. Its cost equals 70 and its similarity equals 348. When reducing redundancy in T_{trad}, we obtain an all-transitions adequate test suite T_{red} with 23 test cases [43], a cost of 58 and a similarity of 270. The manually generated adequate test suite has two test cases [43] for a cost of 31 and a similarity of zero. These are to be compared with all-transitions adequate test suites generated by our GA (TSga1 to TSga4 on the y-axis in Fig. 3): their cost ranges from 46 to 52; their similarity ranges from 0 to 6 (Fig. 3). Without optimizing for similarity (i.e., with only coverage and cost as objective functions), the cost is as low as 39 (Fig. 4).

We first discuss T_{trad}, T_{red} and our GA solutions. A cost of 39 is almost half of the cost obtained with T_{trad} and much less than T_{red}. Comparing these makes sense since similarity is not accounted for when creating T_{trad} and T_{red}. Even when accounting for test case similarities, which we know increases costs, we obtain lower values (46 to 52 instead of 58 or 70).

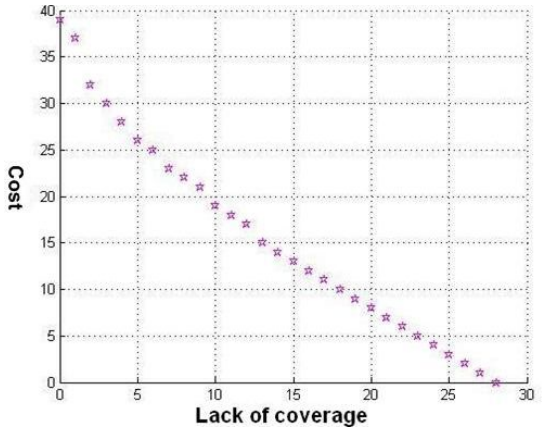

Fig. 4. Cruise control: contents of Pareto front when only optimizing for cost and coverage

With respect to similarity, we observe that the structure of the EFSM graph demands that longer paths than those in T_{trad} or T_{red} be used to increase dissimilarity: this explains why our GA performs much better in terms of similarity (0 to 6 instead of 348 and 270 for T_{trad} and T_{red}). For instance, test paths of length two that cover transitions T_8 or T_{14} all start with T_5 and to make them dissimilar one needs to take some loops on state Inactive/Idle before going to state Active/Running through T_5. We therefore conjecture that, creating a test suite one test case at a time (as in T_{trad}) while ensuring we have dissimilar test cases, would lead to a test suite that is much more expensive than 70, thereby increasing the difference with our approach. We therefore conclude that creating an adequate test suite one test case at a time, each test case satisfying one test objective, is sub-optimal.

When comparing T_{man} with our GA solutions, we observe striking differences in terms of cost and similarity. We tried to understand why our GA is not able to find such a good solution by itself and we therefore determined the size of the search space. We limited the study of the search space to solutions similar to T_{man} in terms of number of test cases (two) and lengths of test cases (15 and 16). Given that each state of the EFSM has seven outgoing transitions, there exist 7^{15} (resp. 7^{16}) different traversals of the EFSM of length 15 (resp. 16). There exist therefore 7^{31} different test suites with the same number of test cases and the same test case lengths as T_{man} in the search space. Considering that these represent a tiny subset of the search space, it is not entirely surprising that our GA does not find such a good solution. We note however that TSga1 is close to T_{man}: (0, 0, 52) instead of (0, 0, 31). Nevertheless, future work will look into better ways to lead the GA to even better solutions than TSga1…TSga4, by for instance revisiting crossover and mutation operators and their probability of occurrence, or by using other objective functions (e.g., a cost measure that accounts for driver/stub/oracle construction might promote a smaller number of test cases, like in T_{man}). We conclude that a human can still do better than our GA, though our GA is very close to a human-generated solution.

We followed the same procedure for Transport Protocol and obtained similar results when comparing our GA solutions to T_{trad}, T_{red} and T_{man} in terms of cost, feasibility and similarity.

E. Results for RQ 2

We verified whether a subset of the test suites in the pareto front for Transport Protocol are indeed feasible: TABLE II. . We managed to manually find parameter values that make each of the test paths in those test suites feasible. Therefore, all the test suites were indeed feasible, despite the seemingly high feasibility values. We may conclude that a high value of our feasibility measure does not necessarily mean that the corresponding test suite is infeasible. Our future work will attempt to identify (possibly rough) thresholds for our feasibility measure which would indicate feasibility is unlikely.

TABLE II. SELECTED TEST SUITES FROM THE PARETO FRONT FOR THE TRANSPORT PROTOCOL CASE STUDY (PATHS IN [43])

Objective Function	Point1	Point2	Point3	Point4	Point5	Point6
Lack of Coverage	0	1	2	3	4	5
Similarity (Levenshtein)	3	4	4	3	2	0
Cost	37	38	34	24	22	20
Feasibility	270	246	198	156	120	108

As an example, referring to the first path of the test suite corresponding to point 1, i.e., Path1: < T_1, T_3, T_{15}, T_8, T_{17}, T_{19}, T_2>, we observe that guard conditions, directly or indirectly, depend on parameter and context variable values. When triggering T_3 we need to select a value for its first parameter that is smaller than the second parameter of T_1 (`opt_ind < prop_opt`) and set its second parameter to an integer greater than zero (`cr > 0`). This is because when T_1 is executed the value of its second parameter is assigned to the context variable `opt` which should be greater than the first parameter of T_3 based on the guard condition of T_3. Restrictions on the second parameter are caused by the guard condition of T_8 (`S_credit > 0`). `S_credit` is a context variable and the last time it is assigned a value before T_8 is executed in the given path is when T_3 is triggered (`S_credit=cr`). The other conditions are on the parameters of T_{15}. The first parameter should be greater than zero (`XpSsq > 0`) and the sum of the two parameters should be either greater than 143 (`XpSsq + cr > 143`) or less than 128 (`XpSsq + cr < 128`). The last two conditions are caused by the guard condition of T_{15} (`TSsq < XpSsq & [cr + XpSsq – TSsq – 128 < 0 V cr + XpSsq – TSsq – 128 > 15]`). `TSsq` is a context variable and the last time it is assigned a value, before T_{15} in the given path, is when T_3 is executed (`TSsq=0`).

It is possible to satisfy all the conditions above and make Path1 feasible (executable). For example, < T_1(3,10), T_3(9,4), T_{15}(10,100), T_8(4,6), T_{17}(), T_{19}(), T_2(10,8,2)> is a feasible path. A complete list of guard conditions, input declarations and transition operations of the Class II transport protocol EFSM can be found in [14].

F. Results for RQ 3

We compared results of our approach with the random test suite generation described in section IV.B.3) in terms of adequacy and feasibility (cost is by design the same as in our GA solutions). We used a typical execution of our GA, which stopped after 151 generations and evaluated 21,350 test suites. We randomly generated the same number of test suites, with the same distribution of overall cost: section IV.B.3).

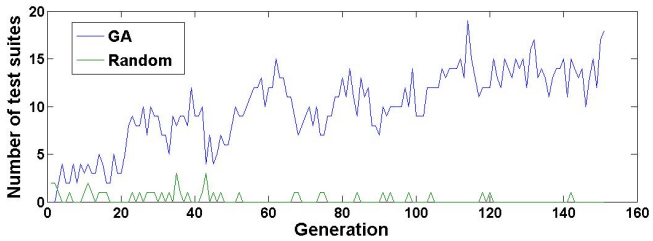

Fig. 5. Number of adequate not absolutely infeasible test paths found at each iteration

Among those two series of test suites, the GA generated series and the randomly generated series, we selected the all-transitions adequate ones, and further selected the ones that have a chance of being feasible. In other words, we discarded test suites that are either inadequate or definitely infeasible. The data flow analysis of Kalaji et al. [14] can identify if a test path in definitely infeasible when for instance the path assigns a constant to a variable and a following guard condition requires the variable to equal another constant. In such a case, the penalty assigned to the path is a very large (simulating infinity) value. Given our feasibility metric, if a test suite has a feasibility value greater than this very large value, we know it contains a test path that is definitely infeasible.

The total number of adequate and not absolutely infeasible test suites generated by the GA was 1,495 while the random generation could only generate 44 such test suites. This means there is a very low chance of achieving our optimization objectives by randomly generating test suites.

Fig. 5 plots the number of adequate not absolutely infeasible test suites found by the GA and the random generation at each iteration of the construction process: random and GA generations occurred concurrently. During the first few iterations, the two procedures are close to each other but in later iterations the number of (adequate, not definitely infeasible) test suites generated by the GA is dramatically higher than the ones randomly generated. This is due to the fact that in early generations, the GA considers costly test suites, and a higher overall cost gives more chances to the random generation to obtain adequate, likely feasible test suites.

G. Results for RQ 4

We selected a couple of adequate test suites generated by our GA and compared them with test suites created by using other testing strategies. For Cruise Control we selected TSga1 and TSga4 (section IV.C) and compared them with four other test suites. We have already explained how T_{man} and T_{red} are created (section IV.D). As mentioned in section IV.B.4) we also used two test suites, using two versions of the round trip path strategy. All the test suites are transition adequate. Mutation score is almost the same for all the test suites: TABLE III. In addition to test suite similarity values with the Levenshtein distance (used by the GA) we also report on the Gower-Legendre measure of similarity since it was suggested as the best measure by others [2]. Our data show a strong

positive relationship between the two: Pearson's r coefficient of .99. Every test suite, except T_{man}, has higher values of similarity and cost compared to ones created by our GA. T_{man}, can be considered as the optimum the GA is searching for. Different types of variability that we have in our encoding (variable number of test cases in test suites as well as variable length test paths) have made it very difficult for the GA to reach that point. However, it has reached points that are much closer to this optimum compared to other approaches.

For Transport Protocol we compared a GA generated adequate test suite (point 1 in TABLE II) to three other test suites: two versions of the round trip path strategy, and one test suite created manually (T_{man}): TABLE IV. The two measures of similarity are once again highly correlated: Pearson's r coefficient of .99. The GA test suite had the highest mutation score. Although T_{man} has the lowest cost it is killing fewer mutants compared to other test suites: TABLE IV. Results of mutation analysis confirmed our intuition about relationship between diversity in a test suite and its cost-effectiveness (TABLE III. and TABLE IV.).

TABLE III. RESULTS OF MUTATION ANALYSIS (CRUISE CONTROL)

	TSga1	TSga4	T_{man}	T_{red}	RTP	RTP (Modified)
Levenshtein	0	6	1	270	296	44
Gower-Legendre	2.74	2.95	0.38	62.85	85.65	10.36
Mutation score	40%	40%	40%	38.38%	38.38%	40%
Cost	52	46	31	58	64	41

TABLE IV. RESULTS OF MUTATION ANALYSIS (TRANSPORT PROTOCOL)

	GA (point 1)	RTP	RTP Modified	T_{man}
Gower-Legendre	1.9347	129.6714	5.6314	1.1833
Levenshtein	3	394	36	5
Mutation score	46.48	40.84	43.38	41.83
Cost	37	89	37	25

V. THREATS TO VALIDITY

Regarding external validity [44], which relates to the extent to which the results of our study can be generalized to other situations, there are two points to consider. First, as mentioned in section IV.B.5), a GA is a stochastic process. Typically, during research, a GA is run a number of times and trends in the results are observed. We ran our GA a number of times on each case study and, although we do not report on trends, we observed results were similar over multiple runs. Therefore, for each case study we report on only one representative run and defer study of trends to the future.

The other external threat is the size and number of case studies. As mentioned in section IV.A we report on two case studies which are admittedly small. But, we believe they are good representatives of what needs to be dealt with in state-based testing. First, similar size state machines are used in UML-based development, and in industrial case studies reported in the literature (section IV.A). Second, it is very uncommon in practice to model an entire system using a single large state machine. Third, these state machines are test models which usually represent a single test purpose [15]. Last, the second case study is a representative sample of a series of case studies used by others [14]. We are currently experimenting with other case study systems.

Internal validity relates to how well we have conducted the studies. Regarding internal validity, feasibility and similarity measures we have used and to what extent they represent feasibility of a test path or diversity in a test suite can be considered as a threat. However, others before us have used exactly the same measures [2], [14] and achieved convincing results. Also as mentioned in section IV.E test paths found by our GA are indeed feasible. We also plan to use other similarity measures in the future.

VI. CONCLUSION AND FUTURE WORK

We proposed a search based technique that generates test suites from an EFSM using a multi-objective genetic algorithm. The goal of our GA is to find test suites that achieve a maximum level of coverage, ideally reaching adequacy (we used the all-transitions selection criterion), have a high chance of being feasible (we use a surrogate measure of feasibility), minimize cost (i.e., cumulative number of triggered transitions in all test cases), and minimize the similarity between the test paths that constitute the test suites since this has been shown to relate to the effectiveness of test cases at finding faults. We provided a detailed description of our algorithm and justified the decisions we made at different steps.

To the best of our knowledge, this is the first time in the state-based testing literature that these objectives are used together. Additionally, our solution creates an entire adequate test suite in one search step instead of creating test cases one after the other for satisfying test objectives (in our case covering transitions) separately in an incremental manner. We argued, and experimentally confirmed, that proceeding incrementally is a sub-optimal procedure for the task, and that this can be compared to a greedy algorithm. Instead, we rely on a meta-heuristic search, specifically a genetic algorithm.

We used two different models to validate our approach. Results confirmed our intuition that generating an entire test suite rather than doing so incrementally results in improvements in terms of cost and dissimilarity, while not hurting with respect to (expected) feasibility and adequacy. Also, we confirmed that when a test suite is expected to be feasible, using a surrogate measure of feasibility, it is indeed possible to find test inputs such that test paths can actually execute, even in the presence of other optimization goals. Also by using diversity as an objective function we managed to find test suites that are less expensive, without scarifying effectiveness at detecting faults.

There is plenty of room to better understand our new technology such as: (a) Investigating different ways of improving the GA itself (e.g., using different probabilities of occurrence of our mutation operators); (b) studying other possible measures to compute coverage (e.g., transition pairs), similarity (see [2]) or cost of a test suite; (c) considering weights for our objective functions, for instance favouring coverage in order to have (hopefully) a larger number of adequate test suites in the Pareto front. The identification of weights is however a difficult problem, and experiments [19] show that a solution with weights does not necessarily perform

better than a real multi-objective solution; (d) trying different strategies for creating the initial population; (e) using other EFSMs (e.g., [14]) to improve external validity; (f) studying the impact of our mutation and crossover operators to improve internal validity.

VII. ACKNOWLEDGMENT

This work was performed under the umbrella of a NSERC-CRD grant, with support from NSERC, CRIAQ, CAE, CMC Electronics, and Mannarino Systems & Software.

REFERENCES

[1] A. Y. Dual and M. U. Uyar, "A method enabling feasible conformance test sequence generation for EFSM models," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 614-627, 2004.

[2] H. Hemmati, A. Arcuri and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM TOSEM*, vol. 22, no. 1, 2013.

[3] M. Harman and B. F.Jones, "Search-based software engineering," *Information and software engineering*, pp. 833-839, 2001.

[4] M. Harman, P. McMinn, J. Teixeira de Souza and S. Yoo, "Search based software engineering: techniques, taxonomy, tutorial," *Empirical Software Engineering and Verification*, vol. 7007, pp. 1-59, 2012.

[5] P. MacMinn, "Search-based software testing: Past, Present and Future," in *IEEE ICST*, Berlin, Germany, 2011.

[6] A. P. Mathur, Foundations of Softwre Testing, Pearson, 2008.

[7] G. Luo, G. Bochmann and A. Petrenko, "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method," *IEEE TSE*, vol. 20, pp. 149-162, 1994.

[8] J. Li and W. Wong, "Automatic test generation from communicating extended finite state machine (CEFSM)-based models.," in *IEEE Int. Symp. on Real-Time Distributed Computing*, 2002.

[9] X. Jin, G. Ciardo, T.-H. Kim and Y. Zhao, "Symbolic verification and test generation for a network of communicating FSMs," in *International conference on Automated technology for verification and analysis*, 2011.

[10] "Conformiq Software Ltd.," [Online]. Available: http://www.verifysoft.com/ttcn-3_qtronic_sip.pdf.

[11] Q. Guo, R. Hierons, M. Harman and K. Derderian, "Computing Unique Input/Output Sequences Using Genetic Algorithms. In: FATES, pp. 164-177 (2004)," in *FATES*, 2004.

[12] J. Tretmans, "Model based testing with labelled transition systems," in *FATES*, 2008.

[13] K. Larsen, M. Mikucionis, B. Nielsen and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study.," in *In: ACM EMSOFT*, 2005.

[14] A. S. Kalaji, R. M. Hierons and S. Swift, "An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models," *Information and Software Technology*, vol. 53, no. 12, pp. 1297-1318, 2011.

[15] M. Utting and B. Legeard, Practical Model-based Testing, Morgan Kaufmann.

[16] C. Schwarzl and B. Peischl, "Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems," in *QSIC*, 2010.

[17] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - A survey," *Proceedings of IEEE*, vol. 84, no. 8, pp. 1090-1123, 1996.

[18] J. Wegener, A. Baresel and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841-854, 2001.

[19] M. Harman, K. Lakhotia and P. McMinn, "A Multi–Objective Approach To Search–Based Test Data Generation," in *GECCO*, London, England, United Kingdom, 2007.

[20] C. C.Michael, G. McGraw and M. A.Schatz, "Generating Software Test Data by Evolution," *IEEE TSE*, vol. 27, no. 12, pp. 1085-1110, 2001.

[21] G. Fraser and A. Arcuri, " Whole test suite generation," *IEEE Transactionaction on Software Engineering*, vol. 39, no. 2, pp. 276-291, 2013.

[22] "Global Optimization Toolbox," Mathworks, [Online]. Available: http://www.mathworks.com/products/global-optimization/. [Accessed 25 April 2013].

[23] C. Coello Coello, D. Van Veldhuizen and G. Lamont, Evolutionary Algorithms for Solving MultiObjective Problems, 2nd ed., Springer, 2007.

[24] J. Horn, N. Nafpliotis and D. Goldberg, "A Niched Pareto Genetic Algorithm for Multiobjective Optimization," in *First IEEE Conference on Evolutionary Computation*, 1994.

[25] D. Goldberg, Genetic Algorithms in Search, Optimization and machine learning, Boston: Addison Wesley, 1989.

[26] S. Silva and E. Costa, "Dynamic Limits for Bloat Control in Genetic Programming and a Review of Past and Current Bloat Theories," *Genetic Programming and Evolvable Machines,*, vol. 10, no. 2, pp. 141-179, 2009.

[27] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Sience and Computational Biology, Cambridge: Cambridge University Press, 1997.

[28] M. Khalil and Y. Labiche, " On the Round Trip Path Testing Strategy," in *IEEE International Symposium on Software Reliability Engineering*, 2010.

[29] L. Briand, Y. Labiche and Y. Wang, "Using Simulation to Empirically Investigate Test Coverage Criteria," IEEE/ACM International Conference on Software Engineering, 2004.

[30] M. Hutchins, H. Froster, T. Goradia and T. Ostrand, "Experiments on the Effectiveness of Dataflow and Controlflow-Based Test Adequacy Criteria," in *IEEE/ACM International Conference on Software Engineering*, 1994.

[31] E. Weyuker, N. Stewart, S. N. Weiss and D. Hamlet, "Comparison of Program Testing Strategies," in *ACM International Symposium in Software Testing and Analysis*, 1991.

[32] P. G. Frankl and S. N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774-787, 1993.

[33] M. Laumanns and E. T. L. Zitzler, "On The Effects of Archiving, Elitism, an Density Based Selection in Evolutionary Multi-objective Optimization," in *International Conference on Evolutionary Multi-Criterion Optimization*, 2001.

[34] R. L. Haupt and S. E. Haupt, Practical Genetic Algorithms, Wiley, 1998.

[35] S. Mouchawrab, L. Briand, Y. Labiche and M. Di Penta, " Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments," *IEEE TSE*, vol. 37, no. 2, pp. 161-187, 2011.

[36] B. Bruegge and A. H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, Prentice Hall, 2004.

[37] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Object Technology, Addison Wesley, 2000.

[38] P. Chevalley and P. Thévenod-Fosse, "Automated Generation of Statistical Test Cases from UML State Diagrams," in *International Computer Software and Applications Conference*, 2001.

[39] N. E. Holt, B. C. D. Anda, K. Asskildt, L. C. Briand, J. Endresen and S. Frøystein, "Experiences with Precise State Modeling in an Industrial Safety Critical System," in *CSDUML*, 2006.

[40] J. H. Andrews, L. Briand, Y. Labiche and A. Namin, "- Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE-TSE*, vol. 32, no. 8, pp. 608-624, 2006.

[41] R. Just, F. Schweiggert and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *ASE*,

Lawrence, KS, USA, 2011.

[42] R. Binder, Testing Object Oriented Systems, Addison-Wesley, 2000.

[43] N. Asoudeh and Y. Labiche, "Multi-objective construction of an entire adequate test suite for an EFSM," Carleton University, SCE-13-05, 2013.

[44] D. Campbell and D. Stanely, Experimental and Quasi-Experimental Designs for Research, Boston, MA: Houghton Mifflin Company, 1963.

VIII. APPENDIX

A. Adequate Test-suites in **Error! Reference source not found.**

TSga1:
Path1: < T_3, T_1, T_6, T_6, T_4, T_5, T_{12}, T_{13}, T_{14}, T_{19}, T_{23}, T_{27}, T_{17}, T_{18}, T_{20}>
Path2: < T_2, T_4, T_5, T_{14}, T_{21}, T_{25}, T_{26}, T_{24}, T_{28}, T_{15}>
Path3: < T_5, T_9, T_{11}, T_{10}, T_9, T_8, T_1, T_7, T_5, T_{11}, T_{14}, T_{16}, T_{18}, T_{15}, T_4, T_5>
Path4: < T_4, T_5, T_{14}, T_{21}, T_{25}, T_{26}, T_{24}, T_{26}, T_{22}>
Path5 :< T_1, T_3>

TSga2:
Path1: < T_6, T_4, T_5, T_{12}, T_{13}, T_{14}, T_{19}, T_{23}, T_{27}, T_{17}, T_{18}, T_{20}>
Path2 :< T_3>
Path3: < T_2, T_4, T_5, T_{14}, T_{21}, T_{25}, T_{26}, T_{24}, T_{28}, T_{15}>
Path4: < T_5, T_9, T_{11}, T_{10}, T_9, T_8, T_1, T_7, T_5, T_{11}, T_{14}, T_{16}, T_{18}, T_{15}, T_4, T_5>
Path5: < T_4, T_5, T_{14}, T_{21}, T_{25}, T_{26}, T_{24}, T_{26}, T_{22}>
Path6 :< T_1, T_3>

TSga3:
Path1 :< T_3>
Path2: < T_6, T_4, T_5, T_{12}, T_{13}, T_{14}, T_{19}, T_{23}, T_{27}, T_{17}, T_{18}, T_{20}>
Path3: < T_2, T_4, T_5, T_{14}, T_{21}, T_{25}, T_{26}, T_{24}, T_{26}, T_{22}>
Path4: < T_5, T_{10}, T_{13}, T_{12}, T_{14}, T_{21}, T_{28}, T_{15}, T_7>
Path5: < T_5, T_9, T_{11}, T_{10}, T_9, T_8, T_1, T_7, T_5, T_{11}, T_{14}, T_{16}, T_{18}, T_{15}, T_4, T_5>
Path6 :< T_7>

TSga4:
Path1: < T_4, T_5, T_{11}, T_{13}, T_{12}, T_{14}, T_{21}, T_{28}, T_{15}, T_7>
Path2: < T_5, T_9, T_8, T_7>
Path3: < T_2, T_4, T_5, T_{14}, T_{21}, T_{25}, T_{26}, T_{24}, T_{26}, T_{22}>
Path4: < T_5, T_{13}, T_{10}, T_{12}, T_{10}, T_{14}, T_{16}, T_{19}>
Path5: < T_5, T_{13}, T_{14}, T_{19}, T_{23}, T_{27}, T_{17}, T_{18}, T_{20}>
Path6 :< T_1, T_3>

B. Test-suites Corresponding to Points in **Error! Reference source not found.**

Point1: adequate test suite
Path1: < T_1, T_3, T_{15}, T_8, T_{17}, T_{19}, T_2>
Path2: < T_2, T_6, T_{14}, T_{18}, T_{21}, T_1, T_5, T_2, T_7, T_{21}>
Path3: < T_1, T_5, T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21}>
Path4: < T_2, T_6, T_{11}, T_{12}, T_9 >

Point2:T_9 is not covered
Path1:< T_2, T_7, T_{21}, T_1, T_3, T_{12}, T_{18}>
Path2: < T_1, T_3, T_{11}, T_{15}, T_8, T_{17}, T_{19}, T_2, T_7, T_{21}, T_1>
Path3:< T_2, T_6, T_{14}, T_{18}, T_{21}, T_1, T_5>
Path4: < T_1, T_5, T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}>
Path5:< T_1>

Point3: T_9 and T_{12} are not covered
Path1: < T_1, T_3, T_{11}, T_{15}, T_8, T_{17}, T_{19}, T_2>
Path2:< T_2, T_6, T_{14}, T_{18}, T_{21}, T_1, T_5, T_2, T_7, T_{21}>
Path3 :< T_1>
Path4: <T_1, T_5, T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21}>

Point4: T_9, T_{12} and T_{14} are not covered
Path1:< T_2, T_6, T_{15}, T_{18}, T_{21}, T_1, T_5, T_3, T_8, T_{11}>
Path2: < T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21}>

Point5: T_8, T_9, T_{12} and T_{15} are not covered
Path1:< T_1, T_3, T_{18}>
Path2:< T_1, T_3, T_{14}, T_{11}>
Path3: < T_1, T_5, T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21}>

Point6: T_3, T_8, T_{12}, T_{14} and T_{15} are not covered
Path1:< T_2, T_6, T_{11}, T_9, T_{18}>
Path2: < T_1, T_5, T_2, T_6, T_{13}, T_{16}, T_{10}, T_{17}, T_{20}, T_1, T_4, T_{19}, T_2, T_7, T_{21}>

C. Test suites used to answer RQ 1

1) T_{trad}

T1	T5-T11	T5-T14-T20
T2	T5-T12	T5-T14-T21
T3	T5-T13	T5-T14-T19-T22
T4	T5-T14	T5-T14-T20-T23
T5	T5-T14-T15	T5-T14-T21-T24
T6	T5-T14-T16	T5-T14-T19-T25
T7	T5-T14-T17	T5-T14-T20-T26
T5-T8	T5-T14-T18	T5-T14-T21-T27
T5-T9	T5-T14-T19	T5-T14-T19-T28
T5-T10		

2) T_{red}

T1	T5-T10	T5-T14-T19-T22
T2	T5-T11	T5-T14-T20-T23
T3	T5-T12	T5-T14-T21-T24
T4	T5-T13	T5-T14-T19-T25
T6	T5-T14-T15	T5-T14-T20-T26
T7	T5-T14-T16	T5-T14-T21-T27
T5-T8	T5-T14-T17	T5-T14-T19-T28
T5-T9	T5-T14-T18	

3) T_{Man}

T1-T2-T3-T4-T6-T7-T5-T8-T5-T9-T10-T11-T12-T13-T14-T15 T5-T14-T16-T17-T18-T19-T23-T28-T20-T26-T27-T21-T25-T24-T22