Automated Generation of State Abstraction Functions using Data Invariant Inference

Paolo Tonella, Cu Duy Nguyen, Alessandro Marchetto
Fondazione Bruno Kessler, Trento, Italy
{tonella, cunduy, marchetto}@fbk.eu

Kiran Lakhotia, Mark Harman
University College London, UK
{k.lakhotia, mark.harman}@ucl.ac.uk

***Abstract*—Model based testing relies on the availability of models that can be defined manually or by means of model inference techniques. To generate models that include meaningful state abstractions, model inference requires a set of abstraction functions as input. However, their specification is difficult and involves substantial manual effort. In this paper, we investigate a technique to automatically infer both the abstraction functions necessary to perform state abstraction and the finite state models based on such abstractions. The proposed approach uses a combination of clustering, invariant inference and genetic algorithms to optimize the abstraction functions along three quality attributes that characterize the resulting models: size, determinism and infeasibility of the admitted behaviors. Preliminary results on a small e-commerce application are extremely encouraging because the automatically produced models include the set of manually defined gold standard models.**

***Keywords*-Model inference; Abstraction functions; Model-based testing; Search-based software engineering.**

I. Introduction

Model-based testing (MBT) is an approach to automate test case generation using a model of the application under test. It has been applied successfully at different levels of testing and in various application domains [4], [13]. One of the key elements of MBT is the *model* that describes the behavior of the system under test (SUT). Such a model is supposed to provide an abstract view of the SUT, by focusing on specific aspects, *e.g.*, the change of a system state at runtime. One of the most frequently used kinds of models is the Finite State Machine (FSM) model. In a FSM model, nodes represent states of the SUT and can be determined, amongst other ways, by the values of class attributes (in case of object-oriented applications [15], [9]), or the values of graphical objects (in case of GUI-based applications [1], [16], [12]). A transition of a FSM represents an event or an action (*e.g.*, a method call, an event handler, etc.) that results in changing the system state.

A FSM of a SUT can be constructed at design time, or inferred from the implementation by means of reverse engineering techniques (*e.g.*, it can be inferred from execution traces). The former case involves manual specification and maintenance effort, which is costly for systems that change frequently. It is also inadequate for systems that can evolve at runtime and where only partial knowledge is available at design time. As a result, most of the recent research effort is devoted to model inference techniques that automatically infer models from execution traces [2], [3], [11], [12].

FSM-based models can be mined by means of two main approaches: event sequence or state abstraction inference. Event sequence abstraction takes advantage of regular language mining algorithms, such as k-tail [2], or its variants [11], [10] to produce a model that generalizes a set of event sequences. State based abstraction maps the concrete (and traced) states into abstract states, and events into transitions between abstract states [3], [12]. The main disadvantage of event sequence abstraction, as compared to state abstraction, is that the resulting FSM contains states that are just placeholders, while the behavior is modeled exclusively by the FSM paths. Obtaining a FSM whose states are meaningful and representative of true states of the SUT is important for many MBT approaches. In this paper, we focus only on state abstraction for FSM inference.

Since the space of concrete states of a system at runtime is virtually unbounded, a state abstraction mechanism to map concrete states to a manageable number of *abstract states* is needed. For instance, Marchetto et al. [12] utilize a set of *abstraction functions*, while Dallmeier at al. [3] use an inspection mechanism that requires implementing *inspector* code. These approaches require manual effort to initially define and later maintain the abstraction mechanisms applied to execution traces in order to obtain the FSM.

In this paper, we propose a novel approach that can automatically infer and refine state abstraction functions and optimize their output model to meet a set of quality attributes. More specifically, we apply a data-clustering algorithm to execution traces with concrete states in order to group concrete states into clusters. We then run invariant inference [5], [6] on each cluster to infer a set of invariants for each cluster, and we iteratively improve the clustering, using a Genetic Algorithm (GA), so as to optimize the quality attributes of the associated FSM model. Each distinct set of invariants produced for each cluster at the end of the optimization represents an abstract state and is used as the abstraction function that maps concrete states to abstract ones. By applying these abstraction functions to concrete input traces we generate the output (FSM) model. To the best of our knowledge, this is the first approach that tries to automatically infer both the abstraction functions and the abstract model from execution traces.

The remainder of this paper is organized as follows: Section II introduces the proposed technique on a running example; Section III provides the algorithmic details of the proposed

technique; Section IV discusses some preliminary experimental data, collected on the running example. The last section is devoted to conclusions and future work.

II. Motivating Example and Definitions

In this section we explain why the problem of defining an abstraction function for state based model inference is intrinsically a multi-objective optimization problem. We do so by resorting to a running example, the *Cart* web application (204 Java LOC; 4 classes). Although *Cart* is an extremely simplified example of a rich-client web application, it holds many of the key traits of existing e-commerce web applications.

Cart provides the typical client side operations implemented to handle the shopping cart of an e-commerce web application. Users can add or remove items to the shopping cart and they can eventually check-out, by moving to a payment procedure. The client side events that trigger such operations are respectively *add*, *rem* and *pay*. The client side state consists of a variable N, holding the number of items currently in the cart. N is assigned the value -1 (*null*) when a user moves to the payment procedure.

TABLE I
FOUR EXECUTION TRACES FOR THE *Cart* APPLICATION.

EVENT	N
START	0
add	1
add	2
rem	1
pay	-1
START	0
add	1
pay	-1

EVENT	N
START	0
add	1
add	2
add	3
add	4
rem	3
add	4
pay	-1

EVENT	N
START	0
add	1
rem	0
add	1
rem	0
add	1
add	2
pay	-1

Table I shows the traces associated with four user sessions of *Cart*. In the first session, the user adds an item to the (initially empty) shopping cart, adds another item, removes an item and then pays for the remaining item. The other user sessions in the table read similarly.

TABLE II
OPTIMAL (TOP) AND SUB-OPTIMAL (BOTTOM) ABSTRACTION FUNCTIONS FOR THE *Cart* APPLICATION.

A1	$f(N) := \{(N \geq 0), (N = -1)\}$
A2	$f(N) := \{(N = 0), (N \geq 1), (N = -1)\}$
A3	$f(N) := \{(N = 0), (N = 1), (N \geq 2), (N = -1)\}$
A4	$f(N) := \{(N = 0), (N = 1), (N = 2), (N \geq 3), (N = -1)\}$
A5	$f(N) := \{(0 \leq N \leq 1), (N \geq 1), (N = -1)\}$
A6	$f(N) := \{(N = 0), (N \neq 0)\}$

Table II shows some examples of *state abstraction functions* that can be applied to the traces of *Cart* to obtain a finite state model of the web application client.

Definition 1 (State abstraction function). *A state abstraction function is a mapping from the space of the concrete states to the powerset of the abstract states: $f : C \rightarrow 2^S$, where C is the set of all concrete state values and S is the set of abstract states.*

The abstraction function $A1$ maps the concrete states of variable N to a subset of the two abstract states $\{S_1, S_2\}$. The first abstract state S_1 is included in the set of abstract states returned by f if N is greater than or equal to zero. S_2 is included if N is -1. Abstraction function $A5$ includes the first abstract state S_1 if N is zero or 1, the second abstract state S_2 if N is greater than or equal to 1, and the third abstract state if N is -1.

Notice that some of the abstraction functions in Table II are *deterministic*, *i.e.*, they always return a singleton set. In other words, they partition the concrete state space. This is true of abstraction functions $A1-A4$ and of $A6$. Abstraction function $A5$ is non-deterministic in that it returns a set of states which include the abstract states S_1 and S_2 when N is equal to 1.

If we apply the six abstraction functions in Table II to a representative set of traces for the *Cart* application, which includes the traces in Table I, we obtain the six finite state models in Figure 1. When $A1$ is used, all concrete states reached after *add* or *rem* transitions satisfy the condition of S_1, $N \geq 0$, but not that of S_2, $N = -1$. Hence, the transitions labeled *add*, *rem* in Figure 1.$A1$ are self-transitions of the abstract state S_1. When transition *pay* is executed, the value of N becomes -1, which is mapped to S_2. Hence the transition from S_1 to S_2 labeled *pay*.

When abstraction function $A5$ is used to process the last trace in Table I, transition *rem* is triggered twice in a concrete state where $N = 1$, resulting in $N = 0$. The source state for this transition is $f(1) = \{S_1, S_2\}$, *i.e.*, either state S_1 or S_2. The target state for this transition is $f(0) = \{S_1\}$, *i.e.*, state S_1 only. Hence, two abstract transitions labeled *rem* must be added to the model, a self transition in S_1 and a transition from S_1 to S_2. As another example, the transition *add* produces a non-deterministic transition from S_1 to both S_1 and S_2 if N is initially zero (this happens initially in all traces of Table I), because $N = 1$ satisfies both abstract states S_1 and S_2.

To complete the finite state model in Figure 1, each concrete transition from the execution traces is mapped to a set of abstract transitions that are represented as labeled edges in the model. Specifically, given the concrete transition $C_1 \rightarrow_e C_2$ from concrete state C_1 to concrete state C_2, triggered by event e, transitions $S_1 \rightarrow_e S_2$ are added to the model for all pairs $\langle S_1, S_2 \rangle \in f(C_1) \times f(C_2)$.

If we compare the quality of the finite state models $A1-A6$ from Figure 1, we can notice that there are several dimensions to consider. First of all the *size*. Some models are smaller and easier to understand, some are larger and more complex. The degree on non-determinism of the model is another important quality attribute. Deterministic models are capable of discriminating the effects of a transition better than non-deterministic models, which admit multiple possible effects. It should be noted that using a deterministic abstraction function does not ensure that the resulting model is deterministic. For instance, the abstraction function $A2$ is deterministic, but the associated model (see Figure 1.$A2$) has one non-deterministic transition, associated with the event *rem* occurring in state S_2. In fact, the concrete state $N = 1$ belongs to S_2, but also

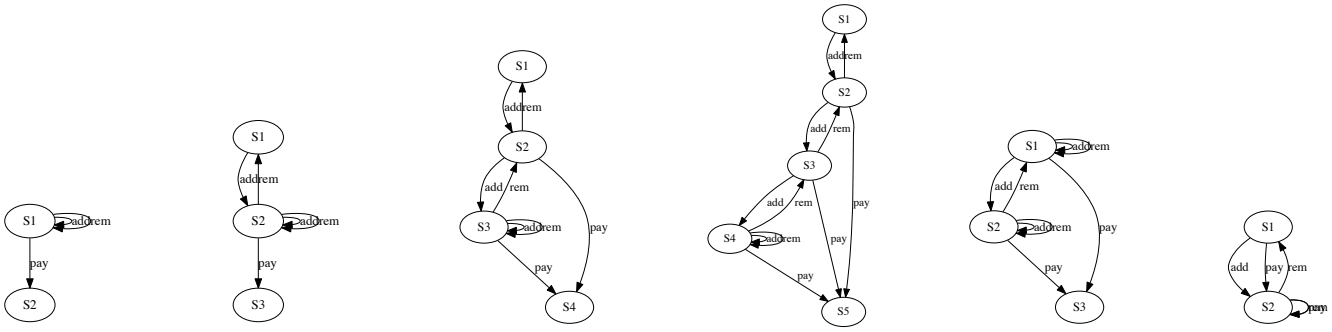

Fig. 1. Finite state models obtained by applying the abstraction functions in Table II.

$N = 2$ (and any concrete value greater than or equal to 1) is mapped to the same abstract state. When *rem* is executed in the first concrete state ($N = 1$), the target state will be S_1, the state returned by $f(0)$, while from the second concrete state ($N = 2$) another abstract state is returned: S_2, the only element of $f(1)$.

Another quality dimension, which is extremely important when models are used for testing, is the degree of *over-approximation* of the model. In fact, the model might admit more behaviors than the system being modeled. This is a major problem for model-based testing, since the infeasible behaviors admitted by the model might be regarded as testing targets which will never be achieved. Determining that they are unachievable targets is undecidable in the general case. Even for the many practical cases in which infeasibility can be decided, the manual effort usually involved in such assessment can be substantial. Model $A1$ in Figure 1 admits several infeasible behaviors, associated with the execution of *rem* in state S_1 when the actual concrete state is $N = 0$. In fact, the item removal operation is disabled in the *Cart* application if no item is present in the cart. Models $A2 - A4$ resolve this issue, since they do not allow any *rem* operation in the initial state S_1, where N is zero.

A. Quality of the finite state models

In this section we introduce three metrics that can be used to characterize the multi-dimensional quality of a finite state model obtained through state abstraction. It should be noted that these metrics are just one way to quantify the intuitive notion of model quality, as introduced in the previous section. Alternative metrics could be defined to capture the same quality attributes. Moreover, our definition of the metrics in this paper is such that lower values are preferred to higher ones. Hence, the model inference problem can be reduced to a multi-objective minimization problem.

For the *size* quality attribute, we use the metric SIZE:

Definition 2 (SIZE). *Size of a finite state model is the number of states that the model has.*

An alternative characterization of this quality attribute might count the number of transitions, instead of the states, or, *e.g.*, the number of independent paths. The simple quality characterization offered by our SIZE metrics has been found to be always adequate for the models investigated in our experiments. The size of models $A1 - A6$ in Figure 1 is (2, 3, 4, 5, 3, 2).

For the quality attribute associated with the degree of *non-determinism*, we use the metric NDET:

Definition 3 (NDET). *NDET of a model is the number of non-deterministic transitions in the model. A transition from a state is non-deterministic if by following it more than one target state is admitted by the model.*

We have a non-deterministic transition whenever it is associated with more than one target state when executed from a given state. For instance, *rem* executed in state S_2 of model $A2$ (see Figure 1) may lead to S_1 or S_2, hence it contributes to an increased value of NDET. Since this is the only non-deterministic transition of $A2$, the final value of NDET for $A2$ is 1. As another example, for $A5$ NDET is equal to two, because of the transition *rem* in state S_2 and of the transition *add* in S_1.

For the quality attribute associated with the degree of *over-approximation* we use the metric INFEAS$_k$:

Definition 4 (INFEAS$_k$). *INFEAS$_k$ of a model is the number of infeasible execution sequences among all possible sequences of length 1 to k generated from the model.*

Since in general the number of infeasible sequences admitted by a model is infinite (see $A1$ in Figure 1 where any sequence of *rem* transitions alone is infeasible), we consider only sequences up to a maximum length k. Once these are generated, we determine whether they are feasible or not by executing them against the system being modeled. If the system does not accept them (*e.g.*, by returning an error), the sequence is regarded as infeasible. Since the number of sequences up to length k increases exponentially with k, in practice relatively small values of k can be used. Of course, keeping k as high as possible given the computational resources available, ensures that the quality of models is

discriminated using a larger sample of sequences. In our experiments we used a value of $k = 7$. Another difficulty with this metric is the assessment of infeasibility for a specific sequence, among those having a length between 1 and k. In fact, the system might reject a sequence in a given execution context (*e.g.*, for a given parameter or input value), but it may accept it in another one. In general, such assessment is undecidable, hence we recognize that in practice this metric can only be approximated. Its precise value will be unknown for arbitrary systems. However, in our experiments we found that the approximations introduced by small values of k, and by the heuristic assessment of infeasibility, do not affect the usability of this metric for the purpose of making comparative evaluations of the quality of alternative models produced by automated state abstraction.

TABLE III
QUALITY METRICS FOR THE FINITE STATE MODELS IN FIGURE 1.

Model	SIZE	NDET	INFEAS_7
A1	2	0	186
A2	3	1	58
A3	4	1	16
A4	5	1	2
A5	3	2	186
A6	2	1	337

Table III shows the values of the quality metrics for the finite state models in Figure 1. Since one model could be better than another model in one dimension, but not in another, we resort to the notion of *dominance* to define a *set* of optimal models (*i.e.*, the Pareto-front in multi-objective optimization). We say model M_1 *dominates* model M_2 ($M_1 \prec M_2$) if along all dimensions M_1 is better than or equal to M_2 (with at least one dimension on which a strict inequality holds):

$$\begin{aligned} M_1 \prec M_2 \quad \textit{iff} \quad & \text{SIZE}(M_1) \leq \text{SIZE}(M_2) \wedge \\ & \text{NDET}(M_1) \leq \text{NDET}(M_2) \wedge \\ & \text{INFEAS}_k(M_1) \leq \text{INFEAS}_k(M_2) \wedge \\ & (\text{SIZE}(M_1) < \text{SIZE}(M_2) \vee \\ & \text{NDET}(M_1) < \text{NDET}(M_2) \vee \\ & \text{INFEAS}_k(M_1) < \text{INFEAS}_k(M_2)) \end{aligned}$$

The Pareto front of the *optimal* models is then defined as the set of non-dominated models. In our running example (see Table III) the following dominance relations hold: $A1 \prec A5$, $A1 \prec A6$, $A2 \prec A5$. Hence, models $A1 - A4$ are non-dominated; when compared between each other, none of them is superior to the other in all dimensions. For instance, $A1$ has a smaller size and number of non-deterministic transitions than all the other models, but it is the worst in terms of infeasible sequences. $A2 - A4$ are equal for what concerns the degree of non-determinism, but while their size increases, making them increasingly more complex, the number of infeasible sequences is correspondingly reduced. Hence, a compromise must be made, in this case between size of the model and number of infeasible behaviors admitted by the model.

III. APPROACH

Our approach aims at automatically generating abstraction functions that can be used for state-based model inference. Abstraction functions are obtained by applying data invariant inference (*e.g.*, Daikon [6]) to the concrete states observed in the execution traces. A *concrete state* is a variable-value mapping observed at a given execution point. In practice, projections of the actual concrete states (instead of complete concrete states) might be traced, for scalability reasons. Concrete states can be represented as tuples of values, where the position in the tuple determines the related state variable. The tuple representation of concrete states is used by the clustering algorithms. For instance, the initial concrete state of *Cart* is always $N = 0$ (tuple $\langle 0 \rangle$), hence the invariant $(N = 0)$ can be easily determined if data invariant inference is applied to the initial state appearing in all available traces. However, in the general case, data invariant inference is applied to a collection (a *multiset*) of concrete states (tuples). Hence, the problem is how to find cohesive groups of concrete states to be considered together during invariant inference.

The problem can be formulated as a *partitioning* problem over the space of the concrete states: *Given a set of concrete states, find a partition such that the invariants obtained from each subset in the partition produce an optimal model when used as abstraction functions*.

The number of possible partitions is exponential with the size of the concrete state space. Hence it cannot be explored exhaustively. Moreover, the relation between a given partition and the optimality of the resulting model is so indirect that no analytical solution can be found. Hence, we turn to Search Based Software Engineering (SBSE) to find an approximate solution to the problem [8], [7]. Specifically we use clustering to produce an initial set of candidate solutions which is then evolved by a multi-objective genetic algorithm.

A. Algorithm

Algorithm 1 shows the key steps for the computation of invariants, associated with clusters of concrete states, and for the optimization of such invariants, in terms of the multi-dimensional quality metrics of the associated models.

Clustering is applied initially (at Step 2), to seed the genetic algorithm used for multi-objective optimization with a reasonably good starting solution. In fact, the space of possible partitions is so large (exponential with the number of concrete states) that a randomly generated partition would require substantially more optimization effort than a solution obtained from clustering, in which cohesive groups of concrete states are already formed. Any clustering algorithm can be used in this step. We experimented with both agglomerative clustering and k-means, as implemented in the Weka[1] data mining library. To obtain a diverse population of initial individuals, clustering parameters have been varied so as to produce multiple clustering solutions. For instance, with k-means this involves varying the value of k; with agglomerative clustering it involves varying the cut level.

In Step 3, each partition produced by clustering is encoded as a sequence of integers. Such a sequence represents the chromosome of the individuals that are evolved

[1] http://www.cs.waikato.ac.nz/ml/weka/

Algorithm 1 Abstraction function inference

Input T: execution traces; each trace $t \in T$ is a sequence of pairs $\langle e, c \rangle$, where e is an event and $c : X \rightarrow V$ is a concrete state, i.e., a variable-value ($X \rightarrow V$) map

Output set of $\{I_1(X), I_2(X), ...\}$: set of optimal state abstraction functions, each a set of boolean predicates defined over variables X

1: Let $C := \langle c_1, ..., c_n \rangle$ be the set of all unique concrete states appearing in T
2: Apply clustering to partition C into the cohesive subsets $C_1, C_2, ...$
3: Encode each partition as a sequence of integers $ch = \langle i_1, ..., i_n \rangle$
4: Seed the initial population *Pop* with the partitions (individuals) ch
5: **while** search budget still available **do**
6: Compute the invariants $I_1, I_2, ...$ for the clusters $C_1, C_2, ...$ of each individual ch
7: Optimize the invariants
8: Generate a model M for each individual ch using its state invariants $I_1, I_2, ...$
9: Compute (SIZE(M), NDET(M), INFEAS$_k$(M)) for each model M
10: Determine the non-dominated models and update the Pareto front
11: Apply genetic operators to evolve the current population *Pop*
12: **end while**
13: Return the Pareto front of optimal state abstraction functions

by the genetic algorithm. In the *Cart* example, we may get the following two partitions from clustering: $P_1 = \{\{N{=}0, N{=}1, N{=}2, N{=}3, N{=}4\}, \{N{=}{-}1\}\}$ and $P_2 = \{\{N{=}0\}, \{N{=}1, N{=}2, N{=}3, N{=}4\}, \{N{=}{-}1\}\}$. Let $C = \langle N{=}{-}1, N{=}0, N{=}1, N{=}2, N{=}3, N{=}4 \rangle$ be the sequence of all concrete states. Then, partition P_1 can be encoded as the following chromosome: $ch_1 = \langle 2, 1, 1, 1, 1, 1 \rangle$, meaning that the first concrete value of C is assigned to the second cluster of P_1, while all the other following values from C are assigned to cluster number 1. Similarly, if we encode P_2 into a chromosome, we get: $ch_2 = \langle 3, 1, 2, 2, 2, 2 \rangle$.

In Step 6 we employ data invariant inference (specifically, Daikon[2]), to obtain a predicate (invariant) that characterizes each cluster of each individual. In our running example, we may obtain invariants $(N \geq 0)$ and $(N = -1)$ for the two clusters of P_1; $(N = 0)$, $(N > 0)$ and $(N = -1)$ for the three clusters of P_2. It should be noted that Daikon must be applied to the *multiset* (where repeated occurrences of the same value are permitted) of concrete states that belong to each cluster, such that the multiplicity of occurrence of each state value is taken into account in the measure of confidence that is internally used by Daikon to filter the candidate invariants.

Based on our experience with the invariants inferred by Daikon for clusters of concrete states, we found it useful to apply the following invariant optimizations (at Step 7):

- Whenever two clusters have the exact same invariants, they are merged and regarded as a single cluster with the given invariant.
- Whenever the invariants I_1, I_2 of two clusters are satisfied by the exact same set of concrete states, the two clusters are merged into one and associated with the invariant $I_1 \vee I_2$.
- If the model generated from the set of invariants $I_1, I_2, ...$ contains pairs of nodes $\langle n_1, n_2 \rangle$ that can be merged according to the k-tail principle [2] (*i.e.*, they cannot be distinguished based on the paths of length k that start from them), the pairs of invariants I_{n_1}, I_{n_2} associated with two such nodes are replaced by $I_{n_1} \vee I_{n_2}$ and the corresponding clusters are merged.

Once invariants are available for the clusters of each individual, we use them as the abstraction functions needed for state-based model inference (Step 8). Concrete states in the available execution traces are mapped to abstract states by determining which invariants they satisfy. Transitions are added to the inferred model for each action reported in the traces. For an action between a source concrete state and a target concrete state, transitions are added between all abstract states satisfied by the source concrete state and all abstract states satisfied by the target concrete state. An example of this construction is reported in Section II for the *Cart* application.

Models for the current population *Pop* are evaluated using the quality metrics SIZE, NDET and INFEAS$_k$ (Step 9). These metrics have been described and motivated in Section II. In Step 10, optimal solutions (*i.e.*, non-dominated solutions) are recorded in the Pareto-front, which is eventually returned by the algorithm (at Step 13).

Step 11 applies genetic operators to evolve the current population of individuals. First, a selection operator is used to produce a new population in which individuals associated with good quality metrics SIZE, NDET and INFEAS$_k$ are preserved, while individuals with poor metric values are discarded. We adopted pairwise tournament selection, in which the dominance relation is used to decide the winner of the tournament. After selection, surviving individuals are mutated and crossed-over, to explore alternative partitions of the concrete state space. We used the following mutation operators:

- **MOVE**: Randomly re-assign a concrete state to a different (existing or new) cluster.
- **MERGE**: Merge two randomly chosen clusters.
- **SPLIT**: Split a cluster into two, by randomly assigning concrete states to either of the two new clusters.

All three operators could be improved by taking into account the distance between concrete states used by clustering. For instance, MOVE may choose the closest cluster for the concrete state being re-assigned, instead of a random cluster. MERGE may select the two closest clusters, instead of two randomly selected clusters. SPLIT may re-apply clustering to the cluster being split, to obtain two or more new sub-clusters. We intend to investigate these variants in our future work.

With respect to crossover, we use a standard single-point crossover. Given our representation of the chromosomes as a sequence of integers, a single-point crossover ensures only valid partitions of concrete states are generated.

The algorithm terminates when the search budget is over (see Step 5). This can be expressed in terms of maximum

[2]http://groups.csail.mit.edu/pag/daikon/

number of fitness evaluations allowed or a time out for the algorithm. Another alternative is to stop the search when no improvement is observed (no change in the Pareto-front) during the last G generations.

IV. Case Study

We experimented with the simple application *Cart*, presented in Section II, where it is used as a running example, for a preliminary validation of our approach. The aim of our experiments was to answer the following research questions:

RQ1 (Role of optimization): *How does the Pareto-front produced by the genetic algorithm differ from the initial solution produced by clustering alone?*

RQ2 (Quality): *What is the quality of the models produced by our approach, compared to manually defined models of the same application?*

The first research question, RQ1, deals with the contribution of the genetic algorithm to the computation of the optimal abstraction functions. The question is whether clustering alone (*i.e.*, clustering followed by invariant inference and invariant optimization) is enough to produce the final abstraction functions returned by the proposed algorithm. To answer this research question we compare the Pareto front available immediately after clustering to the one returned by the algorithm at the end of its execution. Specifically, we measure the number of dominated solutions in the initial Pareto front and the number of new solutions in the final Pareto front.

The second research question involves a user-centric assessment of the quality of the returned models. The question is whether a manually defined Gold Standard (GS) model for the subject application is similar to, or substantially different from the optimal models produced by the proposed algorithm. To answer this question, three of the authors of this paper agreed on the abstraction functions for the subject application (to avoid any bias, this was done before running our algorithm). The GS FSMs for Cart are reported in Figure 1, models $A1-4$. After executing the abstraction function inference algorithm, we determined the position of the GS FSMs with respect to the Pareto front produced by the algorithm, to see if the GS belongs to the Pareto-front, i.e., whether the proposed approach can produce the manually defined models by automatically inferring the abstraction functions.

A. Experimental Setup

For the validation study carried out in this paper we used the agglomerative hierarchical clustering algorithm available in Weka. In order to obtain a diverse set of initial clusters with which to seed the genetic algorithm, we ran Weka 10 times, varying the number of output clusters so that we ended up with 10 different clusters of concrete states. Each cluster was generated using the following Weka settings: Number of clusters, $-N$, one of $[2, ..., 10]$, link type $-L = Single$, distance function $-A = weka.core.EuclideanDistance -R\ first-last$.

The GA was then configured to maintain a population of 100 candidate solutions. After each generation 10 parents were select for reproduction using a pairwise tournament selection. We applied a one-point crossover operator to the selected parents which produces 10 offspring solutions. Each offspring then had a 60% chance of being mutated. We randomly applied one of the three mutation operators described earlier. As described in Step 7 of Algorithm 1, we merge nodes that share the same k-tail as an additional optimization step. We chose k to be equal to 2 when merging nodes.

Our GA uses an elitist re-insertion strategy for updating the current population to the next generation. If an offspring dominates an existing member of the population (according to our quality metrics), then the old member of the population is replaced by the offspring. We also maintain a separate archive (*i.e.* the pareto front) for all non-dominated solutions the GA found during the duration of the search.

The GA was allowed to run for a maximum of 100 generations. If the Pareto front had not been updated in more than 20 generations (*i.e.* the search is likely to have stagnated), then the search was terminated. We repeated the GA runs 30 times, using different random number seeds, in order to reduce the risk of obtaining chance results.

B. Results and Discussions

We first investigated the quality of the models generated by clustering alone. Initially every cluster corresponds to a state in the FSM. The FSM is then further optimized as described in Step 7 of Algorithm 1. Table IV shows the attributes of these optimized FSMs according to our three metrics.

TABLE IV
Results for RQ1-part 1: This table shows the different models and their quality attributes generated by clustering alone.

$SIZE$	$INFEAS_7$	$NDET$
2	2137	4
3	2137	12
4	2137	24
5	679	40
6	679	56
7	450	78
8	58	63
9	58	89
10	14	119

TABLE V
Results for RQ1-part 2: This table shows a comparison between models generated by clustering in combination with a GA and clustering alone.

Type	Avg. solutions on Pareto front	Avg. $SIZE$	Avg. $INFEAS_7$	Avg. $NDET$
Clustering	0	6.00	927.67	53.89
Clustering + GA	3.13	3.03	81.73	1.13

One problem with using only clustering to generate the FSMs is that one has to decide the number of clusters *a priori*. If too few clusters are specified, the resulting model generalizes the behavior of an application too much. Thus, it is possible to generate many infeasible sequences from such models. This is illustrated by the second column in Table IV.

On the other hand, if too many clusters are used to generate a model, the models tend to exhibit more non-deterministic behavior. As can be seen (from Table IV), the number of non-deterministic sequences in a model increases as the size of the model increases.

We next looked at how the models obtained from clustering compare to models generated by Algorithm 1. Table V shows that clustering alone is not sufficient to generate partitions such that the invariants obtained from each subset in the partition produces an optimal model when used as an abstraction function. None of the solutions generated through clustering alone are on the Pareto-front of models for our *Cart* application. In contrast, if we combine clustering with a genetic algorithm, our technique is able to find 4 models on the Pareto-front. These models correspond to models $A1$ to $A4$ in Figure 1. The models found by the genetic algorithm have a very low level of non-determinism (*i.e.* 1.13 sequences on average). The number of infeasible sequences that can be generated from these models is also low compared to models of similar size (*e.g.*, rows $1-3$ in Table IV) found by clustering alone.

TABLE VI
RESULTS FOR RQ2: OCCURRENCE OF THE GS MODELS $A1-4$ IN THE PARETO FRONTS PRODUCED BY THE PROPOSED TECHNIQUE

GS Model	Pareto fronts including the GS
$A1$	100%
$A2$	86.67%
$A3$	26.67%
$A4$	100%

Table VI shows the percentage of Pareto fronts produced by the proposed approach that contain models defined as GS models for the Cart application. The non-dominated model $A3$ is produced relatively infrequently by our algorithm because the mutation and crossover operators we are using give it a very low chance of being generated. We plan to investigate further and improve our genetic operators in the future. The capability of the proposed technique to produce exactly the same FSM models that an expert would define for the Cart application is very encouraging and indicates that automated inference of abstraction functions has the potential to bridge the gap between model construction and MBT.

V. CONCLUSIONS AND FUTURE WORK

We have presented an approach to generate state abstraction functions from clusters of concrete states and to optimize the FSM models generated from such functions. Optimization is carried out by a multi-objective genetic algorithm that takes into account three quality attributes of the models, size, non-determinism and infeasibility.

On a small e-commerce application, Cart, the proposed algorithm was able to generate exactly the same FSM models that some of the authors defined as gold models for the target application. Such models are not trivial to obtain automatically, since clustering of the concrete states alone resulted always in sub-optimal models. These results are quite encouraging, indicating that the approach has potential and is viable.

As future work we will conduct additional experiments on larger applications. We will also investigate alternative clustering algorithms and different genetic operators used by the multi-objective optimization.

ACKNOWLEDGEMENT

Mark Harman, Kiran Lakhotia, Alessandro Marchetto, Cu Duy Nguyen and Paolo Tonella are funded through the EU project FITTEST (ICT-2009.1.2 no 257574).

REFERENCES

[1] A. Andrews, J. Offutt, and R. Alexander, "Testing Web Applications by Modeling with FSMs," *Software and System Modeling, Vol 4, n. 3*, pp. 326–345, 2005.

[2] A. Biermann and J. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. on Computers*, vol. 21, no. 6, 1972.

[3] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *Proc. of the International Workshop on Dynamic Analysis (WODA)*, Shangai, China, May 2006, pp. 17–24.

[4] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies (in co-location with ASE)*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 1–25, 2001.

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[7] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, p. Article 11, November 2012.

[8] M. Harman, P. McMinn, J. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification: LASER 2009-2010*, B. Meyer and M. Nordio, Eds. Springer, 2012, pp. 1–59, LNCS 7007.

[9] Y. Kim, H. Hong, D. Bae, and S. Cha, "Test cases generation from uml state diagrams," *Software, IEE Proceedings* -, vol. 146, no. 4, pp. 187 –192, aug 1999.

[10] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic, "Using dynamic execution traces and program invariants to enhance behavioral model inference," in *proceedings of the International Conference on Software Engineering - NIER Track*, 2010.

[11] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *proceedings of the International Conference on Software Engineering*, 2008.

[12] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *Proc. of IEEE International Conference on Software Testing (ICST)*, Lillehammer, Norway, April 2008, pp. 121–131.

[13] M. Shafique and Y. Labiche, "A systematic review of model based testing tool support," Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04, May 2010.

[14] P. Tonella, A. Marchetto, C. D. Nguyen, Y. Jia, K. Lakhotia, and M. Harman, "Finding the optimal balance between over and under approximation of models inferred from execution logs," in *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 21–30.

[15] C. D. Turner and D. J. Robson, "The state-based testing of object-oriented programs," in *Proc. of the Conference on Software Maintenance (ICSM)*. Montreal, Canada: IEEE Computer Society, September 1993, pp. 302–310.

[16] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback to generate test cases," in *Proc. the International Conference on Software Engineering (ICSE)*. Washington, DC, USA: IEEE Computer Society, May 23–25, 2007, pp. 396–405.