Finding the Optimal Balance between Over and Under Approximation of Models Inferred from Execution Logs

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

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

***Abstract*—Models inferred from execution traces (logs) may admit more behaviours than those possible in the real system (over-approximation) or may exclude behaviours that can indeed occur in the real system (under-approximation). Both problems negatively affect model based testing. In fact, over-approximation results in infeasible test cases, *i.e.*, test cases that cannot be activated by any input data. Under-approximation results in missing test cases, *i.e.*, system behaviours that are not represented in the model are also never tested. In this paper we balance over- and under-approximation of inferred models by resorting to multi-objective optimization achieved by means of two search-based algorithms: A multi-objective Genetic Algorithm (GA) and the NSGA-II.**

We report the results on two open-source web applications and compare the multi-objective optimization to the state-of-the-art KLFA tool. We show that it is possible to identify regions in the Pareto front that contain models which violate fewer application constraints and have a higher bug detection ratio. The Pareto fronts generated by the multi-objective GA contain a region where models violate on average 2% of an application's constraints, compared to 2.8% for NSGA-II and 28.3% for the KLFA models. Similarly, it is possible to identify a region on the Pareto front where the multi-objective GA inferred models have an average bug detection ratio of $110:3$ and the NSGA-II inferred models have an average bug detection ratio of $101:6$. This compares to a bug detection ratio of $310928:13$ for the KLFA tool.

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

I. Introduction

Model inference has been successfully employed in many areas, including program comprehension, software testing and evolution [5], [15], [14]. Models can be inferred by means of state abstraction or event sequence abstraction. In state-based abstraction, abstraction functions are defined to map concrete states into abstract states, so as to control the size of the inferred model and to allow for generalization from the actually observed states [5].

Event sequence abstraction takes advantage of regular language inference algorithms, such as k-tail [2], or its variants [11], [13]. A finite state machine is obtained which recognizes the language of the event sequences observed in execution logs. Such finite state machines are actually a generalization of the observed sequences and not just their union. However, these generalizations are usually *unsound*, which means:

- they might introduce infeasible behaviours (paths allowed in the model that are impossible in the real application), hence over-generalizing;
- they might exclude some possible behaviours (paths allowed in the real application that do not exist in the model), hence under-generalizing.

The aim of this paper is to infer models from execution traces for the purpose of software testing. The inferred models will be used to generate abstract test cases[1]. These test cases need to obey certain properties, thus placing the following requirements on inferred models: Test cases generated from a model must be valid, meaning that they represent a feasible execution sequence for an application. Second, it should be possible to generate test cases from the model that exercise parts of an application for which no execution traces exist. Conversely, an inferred model should also accept new, previously unseen execution traces, without having to be updated first. Finally, test cases generated from a model should have the potential to reveal bugs in an application.

In order for a model to fulfill these requirements, it has to successfully balance the over- and under-approximation of an applications' behaviour. In one extreme, under-approximation may be a problem for testing because the model does not contain important behaviours of an application. It simply represents observed execution traces without any generalization.

In the other extreme, over-approximation may also be troublesome for testing, since it can result in invalid test cases. Too much over-approximation gives rise to the generation of event sequences (*i.e.*, test cases) that represent infeasible paths through an application. This is due to models abstracting too much from the observed behaviours.

Over and under-approximation are contrasting properties of inferred models. Therefore, we propose to use multi-objective algorithms to find a good trade-off between models that over- and models that under-approximate the behaviour of a system.

The primary contributions of this paper are:

- We define three metrics that can be used to characterize

[1]An abstract test case is an event sequence defined on the model. To turn it into a concrete test case, input data must be provided.

TABLE I
EXAMPLE EXECUTION TRACES.

T1	T2	T3
println	println	println
Formatter	Formatter	Formatter
format	format	close
close	format	println
println	format	
	close	
	println	

T1,T2: Execution traces used to infer the model shown in Figure 1. *T3: Newly added execution trace.*

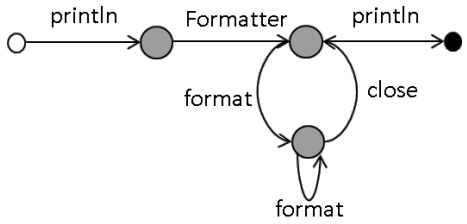


Fig. 1. Model inferred from the execution traces T1,T2 from Table I.

the level of over- and under-approximation in model inference.

- We introduce two multi-objective algorithms, a Genetic Algorithm and the NSGA-II [6], to infer models from execution traces.
- We analyse the inferred models with respect to
  - the trade-off between over- and under-approximation;
  - their validity with respect to obeying application constraints;
  - their potential bug finding ability;
- We evaluate our inferred models against a state-of-the-art benchmark technique (KLFA [16]).

The rest of this paper is organized as follows: The next section provides background information on the problem of balancing over- and under-approximation and introduces three metrics that capture these properties. In Section III we introduce two multi-objective algorithms for model inference, along with a fitness function that operates on the metrics defined in Section II. Section IV presents an evaluation of our proposed inference algorithms, and Section V describes related work. Finally, Section VI draws conclusions and presents future work.

II. OVER AND UNDER APPROXIMATION

Let us consider the model shown in Figure 1 and let us assume it has been inferred from the execution traces `T1` and `T2` from Table I. Now we consider a new execution trace, `T3`, obtained from the same application. Since `T3` represents a valid observed behaviour we would expect our model to accept this trace. In our example however, we see that `T3` is not accepted by the model in Figure 1, because the `close` transition cannot be executed after the `Formatter` transition. Hence, the model is under-generalizing the possible legal behaviours of the application by not admitting a legal execution trace.

On the other hand, we may use the model to generate the following event sequence:

ES1: ⟨`println`, `Formatter`, `format`, `close`, `format`, `close`, `println`⟩

This event sequence is unlikely to represent a legal behaviour of the real system since it involves a double invocation of the events `format`, `close`.

If the model shown in Figure 1 is used to generate abstract test cases, a legal behaviour which excludes the `format` event (see trace `T3`), can never be tested. Yet it is possible to generate infeasible (abstract) test cases such as *ES1*. Hence, over- and under-approximation are a problem for model based testing. We therefore propose a set of metrics to quantify these two properties so that we may use multi-objective algorithms to find good trade-offs between them.

We can assume that the amount of over-approximation (*i.e.*, behaviours that are not possible in reality) is proportional to the number of unobserved event sequences generated from the model (up to a given, maximum length L). We say that an event sequence is unobserved if it is not contained in the set of execution traces obtained from the real system.

For the amount of under-approximation (*i.e.*, behaviours that are possible, but are not accepted by the model) we assume it is proportional to the number of unrecognised traces. A trace is considered unrecognised if it is not accepted by the model. Thus, we consider:

$$\textit{Over-approximation} \sim \textit{UnObs}$$
$$\textit{Under-approximation} \sim \textit{UnRec}$$

However, minimizing the amount of over- and under-approximation is not enough. Consider a model that consists of the union of linear event sequences, corresponding to the execution traces `T1, T2, T3` from Table I. Such a model is shown in Figure 2 (this model is non-deterministic, but it can be easily transformed into an equivalent deterministic model, using the powerset construction algorithm). Since this model reproduces all the execution traces exactly, both the *UnObs* and *UnRec* metrics will be zero.

Based on these values for *UnObs* and *UnRec*, one might wrongly regard such a model as optimal. The problem is that this model does not generalize the observed behaviours in any way. Instead it explicitly represents each and every observed behaviour. Consequently the model is in fact under-approximating (even though it is truly not over-approximating) the behaviour of a system.

If we construct a model using the union of all execution traces, such that *UnRec* and *UnObs* are zero, we notice that the model tends to have many states. Hence, we can use the size of a model as an approximate measure of the lack of generalization performed by the model over the traces. A properly generalizing model will have a smaller size than a model which represents all traces in parallel.

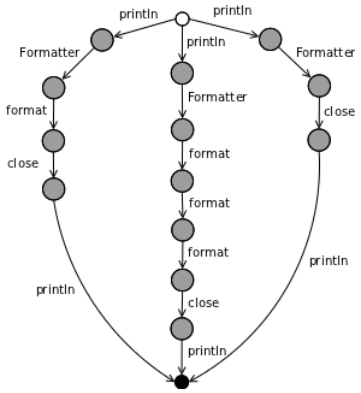


Fig. 2. Non-deterministic model corresponding to the union of execution traces `T1,T2,T3` from Table I.

Therefore, when trying to find a good trade-off between over- and under-approximating models, we include a metric for the degree of "lack of generalization", *i.e.*, *specificity* of the model with respect to the execution traces from which it was inferred. Hence we introduce the following measures:

- *UnObs*: This is a count of how many event sequences (of maximum length L), that do not correspond to any existing execution traces, can be generated from a model.
- *UnRec*: This is a count of how many execution traces are not accepted by a model.
- *Size*: This is a count of the number of states within a model.

We propose to use multi-objective algorithms to optimize (*i.e.*, minimize) those conflicting metrics.

III. Multi-Objective Optimization Of Model Inference

Evolutionary Algorithms [10] are a popular choice for solving multi-objective optimization problems and in this paper we consider two variants: A multi-objective Genetic Algorithm and the NSGA-II [6]. Rather than producing a single model, these algorithms generate a set of *nondominating* (*i.e.*, Pareto optimal) models, which we call the *Pareto front*. We say that a model M_1 dominates another model M_2 if, and only if, M_1 is better than M_2 in at least one objective, and no worse in all other objectives.

Given our three objectives that we want to minimize, we can define the *dominates* function as:

$$
\begin{aligned}
&dominates(M_1, M_2) = \\
&UnObs(M_1) < UnObs(M_2) \wedge UnRec(M_1) \leq UnRec(M_2) \wedge \\
&Size(M_1) \leq Size(M_2) \vee \\
&UnObs(M_1) \leq UnObs(M_2) \wedge UnRec(M_1) < UnRec(M_2) \wedge \\
&Size(M_1) \leq Size(M_2) \vee \\
&UnObs(M_1) \leq UnObs(M_2) \wedge UnRec(M_1) \leq UnRec(M_2) \wedge \\
&Size(M_1) < Size(M_2)
\end{aligned}
$$

Figure 3 shows an example of dominance between two models inferred from all the traces in Table I. *UnObs* was determined by generating all event sequences up to length $L = 4$ and checking whether these are prefixes of actually observed traces. The size of these two models, measured as the number of states, is the same (*i.e.*, 5), but the model on the left has a lower *UnObs* and *UnRec* count.

The *dominates* function is used by both algorithms to guide the search process. The algorithms start by generating an initial population of models. Each model corresponds to a single execution trace and represents a linear sequence of states, with each state (except the start and final state) containing one incoming and one outgoing edge. The population size for the algorithms is fixed at 1000 models. If there are fewer than 1000 execution traces, randomly selected traces are duplicated until enough models can be generated to fill the initial population.

A. Multi-objective Genetic Algorithm

The multi-objective GA uses binary tournament selection to select parents for reproduction. Two models are picked at random from the current population. If one model dominates the other (according to the $dominates$ function), it is added to a mating population. If neither model dominates each other, one of the models is selected at random and placed into the mating population. This process is repeated for all winners of a tournament until the mating population has been filled.

Once the mating population is full, two models are picked at random (until the mating population is empty) to participate in a crossover operation. Each crossover of two models produces a single offspring model. Details of the crossover operator are provided in Section III-C.

After crossover, offspring models are subject to mutation (see mutation operator in Section III-C). The final step of the multi-objective GA is to update the current population with newly generated offspring. For this we use an elitist re-insertion strategy. An offspring model is only accepted into the population if it dominates an existing member of the population. Whenever an offspring model is accepted into the population, the model it dominates is removed, thus keeping the population size constant.

The multi-objective GA also maintains an archive of non-dominated models. These form the (current) Pareto front. At every generation, all nondominated models are copied from the population into the archive. Existing archive members are removed if they are dominated by newly added models.

B. NSGA-II

NSGA-II is an elitist Nondominated Sorting Genetic Algorithm. The algorithm starts by forming an offspring population, using binary tournament selection, crossover and mutation operators as described for the multi-objective GA in the previous sub-section. Successive generations then use an elitist re-insertion strategy to update the parent population. This is done by combining both parent and offspring populations, ranking the combined population, and selecting the top N models as the new parent population.

The ranking of models within the combined population is achieved as follows. First, all models are divided into frontiers. The first front contains only models that are not dominated by any other model. It represents the current Pareto front. The

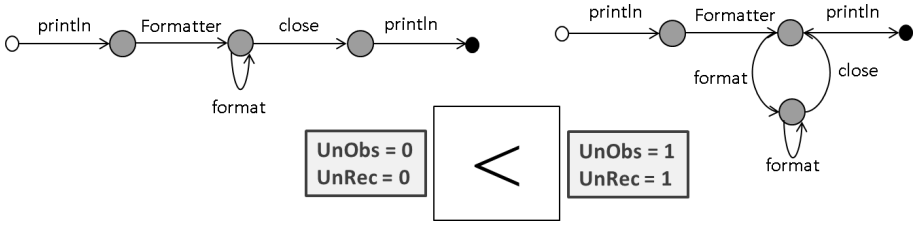


Fig. 3. Dominance example between two inferred models (computed with $L = 4$) having the same size (5 states).

second front contains all models that are dominated by at most one other model, while the third front contains all models that are dominated by at most two models, and so forth.

Once models have been assigned to a frontier, every model within a specific front is assigned a *crowding distance*. The crowding distance measures the average distance of a model, compared to its two neighbouring models within the front, along each of the objectives (*UnObs*, *UnRec* and *Size*). Given two models within the same frontier, one model is preferred over another if it has a greater crowding distance. It means the model lies in a less crowded region of the Pareto front, helping the optimization to create a more diverse set of solutions.

All models can then be ranked according to the frontiers they appear in, with models within the same front sorted according to their crowding distance. Once the parent population has been updated using elitism, a new offspring population is formed in the same way as described for the multi-objective GA.

C. Search Operators

The search space where the Pareto optimal models can be found is the space of all models that can be inferred from the execution traces. In order to explore such a huge space, we define a set of search operators that are analogous to the traditional crossover and mutation operators used in evolutionary computation.

The type of search operators we can define for crossover and mutation operations depends on the abstraction mechanism used by the model inference algorithm. Models can be inferred using an event sequence abstraction and state-based abstraction. In this paper we only consider models inferred using event sequence abstraction.

1) Crossover Operators: Unlike traditional crossover operators used in Evolutionary Algorithms, the crossover operators defined in this paper only produce a single offspring model. We consider two types of crossover operation: *Union* and *Intersection*. For every pair of parent models, we randomly select which operator to apply.

Union: The purpose of the union operator is to reduce the number of unrecognised traces (*UnRec* count) by combining two models M_1 and M_2. The union operator introduces a new unique start node, U_1, for the united model. Then, all nodes from M_1 and M_2 are added in parallel to the new model. Finally, each transition from the start nodes of M_1 and M_2 is added to the new start node U_1. Since the start nodes are likely to share a transition, the new model is potentially nondeterministic. Therefore, we apply powerset construction to make it deterministic again. Figure 4 shows an example of two models and how they are combined.

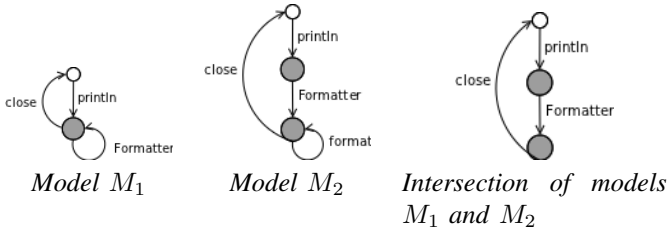


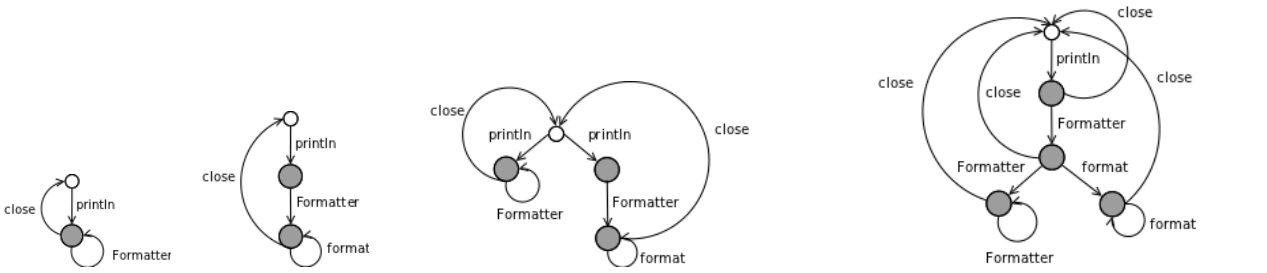
Fig. 5. The right-hand column shows the result of intersecting the two models on the left (M_1) and in the middle (M_2).

Intersection: While the union operator was designed to reduce the number of unrecognised traces, the intersection operation aims to reduce the number of unobserved traces that can be generated from a model (*UnObs* count). The operator starts by creating a unique start node I_1. Then, all transitions of the models, starting with the start nodes, are traversed in parallel in a depth-first manner. At every point, all transitions that are shared between the models M_1 and M_2 are added to the intersected model. Two transitions are considered equivalent if they share the same name.

Figure 5 contains an example of the intersection operator (right-hand column) when applied to the two models in the left columns. Starting at states S_1, both models share the `println` transition. After executing this transition, both models, M_1 and M_2 are in their respective state S_2. From this state both models share the `Formatter` transition. For model M_1 this is a self-transition, thus it remains in state S_2. Model M_2 however is now in state S_3. These two states only share the `close` transition, which takes both models back into their respective state S_1. The right-hand column in Figure 5 shows the model after the intersection operator has been applied. Once again powerset construction is used to make any resulting model deterministic if necessary.

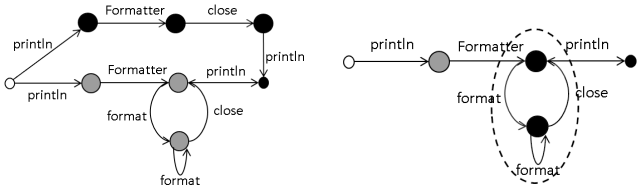
2) Mutation Operators: Every offspring model has a 50% chance of being mutated. We have implemented two mutation operators: *Add Trace* and *Merge States*. As with crossover, we randomly choose which mutation operator is applied to a model.

Add Trace: This operator randomly selects a trace file from the set of execution traces used to generate the initial population. The trace is then added to the existing model. Figure 6a shows how a trace (⟨`println`, `Formatter`, `close`,



Model M_1 *Model M_2* *Nondeterministic union of models M_1 and M_2* *Union of models M_1 and M_2 after powerset construction*

Fig. 4. The right-hand column shows the result of the union of the two models M_1 and M_2.



a) *b)*

Fig. 6. Add trace and merge states operators.

`println`⟩) is added to the start node of the model. The effect of this operation is the same as if an offspring model and a model from the initial population would be combined through the union crossover described earlier.

Merge States: The merge state mutation comes in two flavours: *random-k-tail* and *random merge*.

The *random merge* mutation randomly selects two states, S_1 and S_2, to merge from the model. First, a new state S' is created. Then, all incoming and outgoing transitions of S_1 and S_2 are copied to S'. Finally, the two states S_1 and S_2 are removed from the model.

In a *random-k-tail* merge, two states are selected at random from a model. If the two states share the same k-tail (*i.e.*, they share all transitions up to length k) they are merged. If two states cannot be merged, another two states are selected at random. This process continues until either two states have been merged, or no more states are left to pick from the model.

We set the length of k to 2 in order to increase the probability of state merges to occur. Figure 6b shows an example of two states that may be merged using a k-tail of 2, because the 2-tails of the lower state (⟨`format`, `format`⟩, ⟨`format`, `close`⟩) are also 2-tails of the upper state. Note that in this example we applied an asymmetric version of k-tail, requiring one set of k-tails to be a subset of the other.

The *random-k-tail* mutation is potentially less destructive; states are only merged if they share a k-tail, and thus this mutation is chosen with a 90% probability over the *random merge* mutation. Both mutation operators use powerset construction to make any resulting model deterministic.

IV. Evaluation

The aim of this section is to evaluate the use of multi-objective Evolutionary Algorithms for model inference from execution traces. We selected KLFA as the benchmark reference technique to use for comparison, since KLFA [16] is a recent development of state-of-the-art techniques for model inference using positive examples. It has been shown to be superior to the k-tail based algorithms for traces obtained from software systems [17]. The three research questions to be addressed in this section are as follows:

RQ1 (Trade off): *What trade-offs between over- and under-approximation are determined by multi-objective optimization and how do they compare with the benchmark model inference technique, KLFA?*

RQ2 (Infeasibility): *Does multi-objective optimization find models with fewer infeasible event sequences than the benchmark? What portion of the Pareto front provides a lower number of infeasible sequences?*

RQ3 (Fault revealing): *Does multi-objective optimization find models with a higher fault detection rate (fraction of event sequences that reveal a fault) than the benchmark? What portion of the Pareto front provides a higher number of faults revealed?*

In order to answer these research questions, we selected two open source web applications. `Tudu` [8] is an online to-do list application based on the Java Spring Framework and AJAX. The application allows users to create and edit tasks, as well as share them with other users. Trace files for `Tudu` were obtained by manually executing the application.

We considered two sets of traces. The first, `Tudu-HL` uses a high level abstraction function to transform concrete execution logs into traces that are suitable for model inference. The second set, `Tudu-LL` uses a lower level abstraction function to generate the trace files used for model inference.

`Cyclos` [22] is an on-line banking system that offers a set of e-commerce and communication tools. It is implemented in Java and uses JSP for the web interface. We manually executed the application to obtain a set of execution traces.

In order to answer research questions RQ2 and RQ3 we also collected a set of constraints and bug traces for each application. Constraints denote precedence relationships between event sequences. For example, a constraint of the form ⟨`Formatter`|`format`::`close`⟩ means the `close` event can only be preceded by either the `Formatter` or `format` event.

They describe sub-machines from a *gold standard* model[2]. If it is possible to construct an event sequence from a model where `close` is preceded by any other event, *e.g.* `println`, we consider the model to violate that constraint. The set of precedence constraints for the test subjects were extracted from the real application through manual code-inspection and domain knowledge.

We created the set of bug traces by examining bug reports from the application's bug repositories. A bug trace has the form ⟨`close`, `println`, `format`⟩ and denotes a sequence of events that lead to an error state in the real application. Thus, if a model accepts a bug trace, it is possible to generate a fault revealing abstract test case from the model. Since the ultimate goal of testing is to find bugs, models that are able to reveal more bugs should be preferred over models that fail to reveal (known) bugs. This is akin to mutation testing, where the ability of a test suite to find seeded bugs is used as an indicator of its ability to find unknown bugs in the future.

Note that neither constraint nor bug information is used during the model generation process. We simply use the constraint and bug trace information to evaluate the models once we have completed the inference process.

Both, the multi-objective GA and the NSGA-II, were allowed to run for 100 generations. Due to the stochastic nature of evolutionary algorithms we repeated the model inference for each test subject six times using a fixed set of random number seeds.

We used the model inference tool KLFA [16] as a benchmark against which to evaluate our proposed techniques. The KLFA tool operates in three stages: Preprocessing raw execution logs into trace files using various analysis and abstraction techniques; inferring a model from the trace files; using the inferred model for failure analysis. We only used the model inference part of KLFA, feeding it the same trace files used by the multi-objective GA and NSGA-II.

The model inference part of KLFA works by incrementally building up a model from a set of traces. The algorithm behind KLFA, named *k*-behaviour [17], incrementally builds a model by processing one trace at a time. The idea behind *k*-behaviour is that recurring input patterns occurring in the input traces should be mapped to the same sub-model (*i.e.*, sub-graph of the automaton). Hence, sub-sequences are identified in the input traces that can be completely or partially recognized starting from a state in the model. The model is then extended to accept the trace fully.

Models are generated by the KLFA algorithm in a deterministic way. Therefore, we only ran KLFA once for each set of traces from our test subjects.

A. Answer to Research Questions

Table II shows the total number of models created by the multi-objective GA and NSGA-II (during the six repeat experiments) for each of the applications studied. The second column contains the number of traces used during the inference process. The table also includes a Reference Pareto Front (RPF) made up from all nondominated models found by either the multi-objective GA, NSGA-II or KLFA.

TABLE II
THIS TABLE SHOWS THE NUMBER OF MODELS CREATED BY THE MULTI-OBJECTIVE GA AND NSGA-II FOR EACH OF THE TEST SUBJECTS. RPF STANDS FOR REFERENCE PARETO FRONT.

Subject	Traces	Algorithm	Models	Models on RPF
`Tudu LL`	15	GA	122	41
`Tudu LL`	15	NSGA-II	70	35
`Tudu LL`	15	KLFA	1	1
`Tudu HL`	125	GA	175	51
`Tudu HL`	125	NSGA-II	351	315
`Tudu HL`	125	KLFA	1	1
`Cyclos`	262	GA	160	119
`Cyclos`	262	NSGA-II	397	291
`Cyclos`	262	KLFA	1	1

In all cases, the model inferred by KLFA is on the RPF. This means the multi-objective GA and NSGA-II failed to generate models that are better in all objectives than the KLFA inferred models. In the remainder of this section we analyse the *quality* of the models found in more detail.

Answer RQ1 (Trade off): The goal of the multi-objective algorithms is to find models that contain good trade-offs between under- and over-approximation of an application's behaviour. Figure 7 shows the distribution of models found by the multi-objective GA and NSGA-II, along the *UnObs* objective, in the form of box plots. Figure 9 shows the distribution of models plotted along the *UnRec* objective, and Figure 8 shows the distribution of models plotted along the *Size* objective. As a point of reference we also included the benchmark model produced by KLFA in each of the figures.

The plots are drawn using the union of nondominated solutions found during the six repeated runs of the experiments. The red line inside the boxes show the median value for an objective, while the end points of the whisker lines denote the lowest and highest values, disregarding outliers[3]. Outliers are shown as red plus signs and they denote extreme regions of a Pareto front.

Looking at Figure 7, both the multi-objective GA and NSGA-II are able to infer models that have a low *UnObs* count, *i.e.*, that are not over-approximating the applications' behaviour. In the best case we do not generate any unobserved event sequences from the models (*i.e.*, $UnObs = 0$).

The figures also show that the models inferred by KLFA have a very high *UnObs* count. Even if we take the furthest outlier (with the highest *UnObs* count) from the multi-objective GA or NSGA-II, it has a lower *UnObs* value than the benchmark model. Thus, both the multi-objective GA and NSGA-II models are less over-approximating than the KLFA benchmark.

Recall that the size of a model can also be used as an approximate measure of the level of generalization performed

[2]A gold standard model exactly represents the true application behaviour. Ideally a model inference technique would be able to infer the gold standard model.

[3]An outlier is a value that is more than 1.5 times the interquartile range away from the top or bottom of a box.

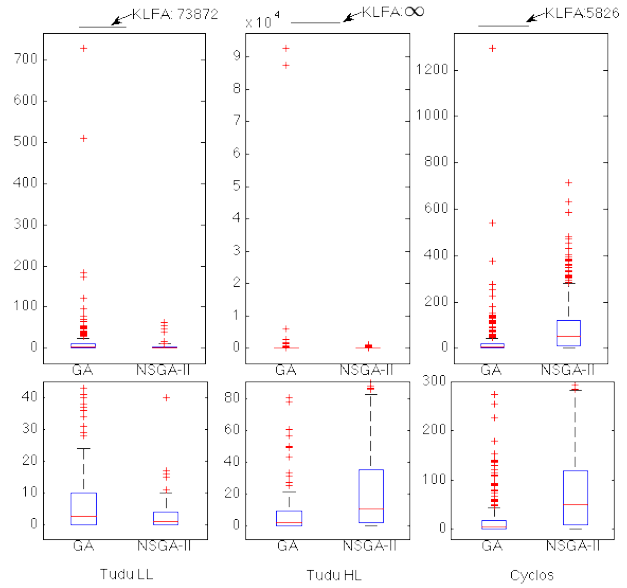


Fig. 7. Box plots of the *UnObs* counts from the models inferred by the multi-objective GA and NSGA-II. The bottom plots zoom in on the box plot regions of the top plots.

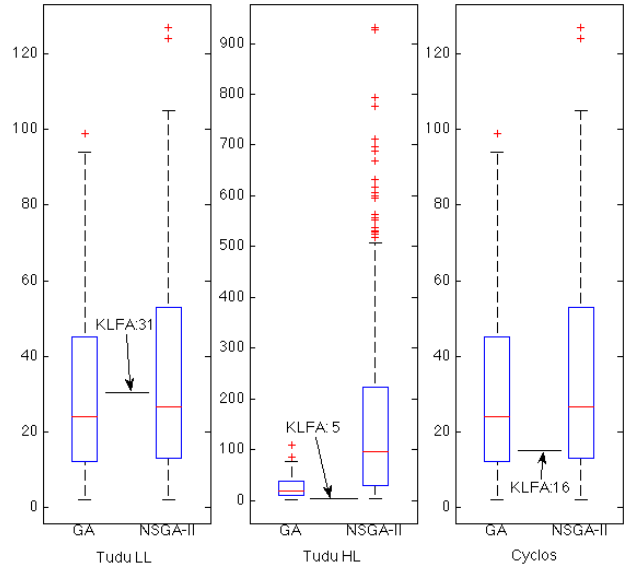


Fig. 8. Box plots of the *Size* counts from the models inferred by the multi-objective GA and NSGA-II.

by the model. Both multi-objective algorithms are able to generate very small models (see Figure 8). In the extreme case, models only contain a single transition. While such models are truly not over-approximating, they are not useful for testing either, because they do not allow us to test the majority of an application's behaviour.

In general, models that have a low *UnObs* count will also be small in size. An exception are models that contain loops. Loops (or self-transitions) enable us to generate infinitely many event sequences while keeping the size of a model small. For example, the KLFA benchmark models are relatively

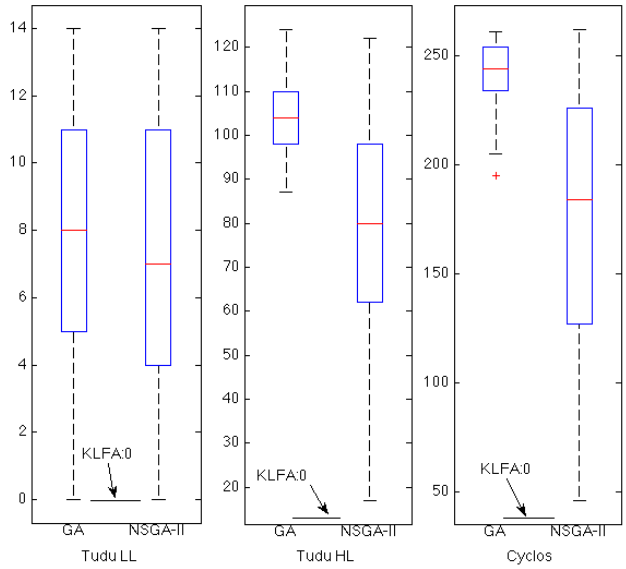


Fig. 9. Box plots of the *UnRec* counts from the models inferred by the multi-objective GA and NSGA-II.

small (see Figure 8), but have a very high *UnObs* count (see Figure 7). Hence they contain a lot of loops.

Finally, Figure 9 shows box plots for the *UnRec* count. The NSGA-II is better at finding models that have a lower *UnRec* value than the multi-objective GA. This means models found by the NSGA-II are less under-approximating.

However, compared to KLFA, both the multi-objective GA and NSGA-II generate worse models in terms of *UnRec*. For all applications and execution traces, the KLFA benchmark has no unrecognised traces. Combined with the data from Figure 7 this is not surprising. The KLFA model tends to over-approximate rather than under-approximate an application's behaviour.

Answer RQ2 (Infeasibility): The motivation behind the work in this paper is to use inferred models for testing, in particular, abstract test case generation. Thus, for models to be useful in practice, they must not violate application constraints. If they do, it is possible to generate invalid test cases, placing extra burden on the tester.

We manually collected a set of constraints for each of the two applications. For `Tudu` we created five and for `Cyclos` 15 constraints. To assess whether a model violates a constraint we examine every state within the model and check if is possible to construct a violating sequence from that state.

Depending on the set of execution traces used for `Tudu`, the model inferred by KLFA violates either four (`Tudu-LL`) out of five or all five constraints (`Tudu-HL`). For `Cyclos`, the KLFA inferred model violates eight of the 15 constraints.

To look at how good the multi-objective GA or NSGA-II inferred models are compared to the KLFA benchmark with respect to *infeasibility* (*i.e.*, constraint violation), we cannot simply pick the best model from a Pareto front and compare it with KLFA's model, as this would be unfair to KLFA.

Multi-objective algorithms do not generate a single solution,

TABLE III
THE COLUMNS SHOW THE TOP 5%, 10%, 50% AND 100% OF MODELS WITH THE LOWEST *UnObs* COUNT AND THEIR AVERAGE NUMBER OF CONSTRAINTS VIOLATED (ALONG WITH THE STANDARD DEVIATION).

Subject	Algorithm	5%	10%	50%	100%
Tudu LL	GA	0.5 (0.8)	1.6 (1.9)	1.8 (1.7)	2.2 (1.7)
Tudu LL	NSGA-II	0.7 (1.2)	1.1 (1.6)	1.6 (1.7)	1.8 (1.7)
Tudu LL	KLFA	–	–	–	4.0 (0.0)
Tudu HL	GA	2.1 (1.6)	2.5 (1.7)	3.1 (1.6)	3.3 (1.5)
Tudu HL	NSGA-II	3.1 (2.1)	3.1 (1.9)	3.8 (1.5)	4.3 (1.2)
Tudu HL	KLFA	–	–	–	5.0 (0.0)
Cyclos	GA	2.4 (2.6)	2.4 (2.2)	4.1 (2.5)	5.3 (2.4)
Cyclos	NSGA-II	4.0 (2.6)	3.9 (2.6)	6.5 (2.4)	8.2 (2.7)
Cyclos	KLFA	–	–	–	8.0 (0.0)

TABLE IV
THE COLUMNS SHOW THE TOP 5%, 10%, 50% AND 100% OF MODELS WITH THE LOWEST *UnRec* COUNT AND THEIR AVERAGE NUMBER OF CONSTRAINTS VIOLATED (ALONG WITH THE STANDARD DEVIATION).

Subject	Algorithm	5%	10%	50%	100%
Tudu LL	GA	4.0 (0.0)	4.0 (0.0)	3.7 (0.7)	2.2 (1.7)
Tudu LL	NSGA-II	4.0 (0.0)	4.0 (0.0)	3.2 (1.2)	1.8 (1.7)
Tudu LL	KLFA	–	–	–	4.0 (0.0)
Tudu HL	GA	4.9 (0.4)	4.6 (0.5)	4.3 (0.5)	3.3 (1.5)
Tudu HL	NSGA-II	4.9 (0.2)	4.9 (0.2)	4.9 (0.3)	4.3 (1.2)
Tudu HL	KLFA	–	–	–	5.0 (0.0)
Cyclos	GA	7.8 (0.5)	7.8 (0.7)	7.2 (1.1)	5.3 (2.4)
Cyclos	NSGA-II	12.2 (1.4)	11.8 (1.2)	10.0 (1.8)	8.2 (2.7)
Cyclos	KLFA	–	–	–	8.0 (0.0)

TABLE V
THE COLUMNS SHOW THE TOP 5%, 10%, 50% AND 100% OF MODELS WITH THE LOWEST *Size* COUNT AND THEIR AVERAGE NUMBER OF CONSTRAINTS VIOLATED (ALONG WITH THE STANDARD DEVIATION).

Subject	Algorithm	5%	10%	50%	100%
Tudu LL	GA	0.2 (0.4)	0.1 (0.3)	0.8 (1.1)	2.2 (1.7)
Tudu LL	NSGA-II	0.0 (0.0)	0.0 (0.0)	0.5 (0.9)	1.8 (1.7)
Tudu LL	KLFA	–	–	–	4.0 (0.0)
Tudu HL	GA	0.4 (0.5)	0.6 (0.5)	2.4 (1.5)	3.3 (1.5)
Tudu HL	NSGA-II	0.6 (0.5)	1.1 (0.9)	3.7 (1.5)	4.3 (1.2)
Tudu HL	KLFA	–	–	–	5.0 (0.0)
Cyclos	GA	0.6 (0.7)	1.4 (1.7)	3.5 (2.0)	5.3 (2.4)
Cyclos	NSGA-II	1.1 (0.9)	2.8 (2.2)	6.5 (2.4)	8.2 (2.7)
Cyclos	KLFA	–	–	–	8.0 (0.0)

but rather a set of solutions. It is up to a decision maker to pick one solution, out of a set of equally good solutions. Therefore, we would like to know from which region of a Pareto front a decision maker should pick a model, and, on average, how many constraints are violated by models from that region.

To answer this question, we generated 3 orderings (see Tables III, IV, V), one for each objective, sorted according to their respective objective values. Then, we took the top 5%, top 10%, top 50% and 100% of models in each ordering and computed how many constraints these models violate.

For example, consider models inferred by the NSGA-II. Table V shows the result for the *Size* ranking. For Tudu-LL, a decision maker can pick any model from the top 5% of models with the lowest *Size* count. Whichever model is chosen, it will, on average, not violate any constraints. In contrast, models inferred by the multi-objective GA violate on average 0.2 constraints, while the model inferred by the KLFA violates four constraints.

The top 5% of models with the lowest *Size* count inferred from Tudu-HL traces violate on average 0.4 (multi-objective GA) and 0.6 (NSGA-II) constraints. Similarly for Cyclos, the top 5% of models with the lowest *Size* count violate on average 0.6 (multi-objective GA) and 1.1 (NSGA-II) constraints. For comparison, KLFA models for these traces violate five (Tudu-HL) and eight (Cyclos) constraints respectively.

Using any other region from the Pareto fronts, *e.g.* top 5% with the lowest *UnObs* or top 5% with the lowest *UnRec*, yields models that violate on average more constraints. Thus, a decision maker should pick a model that lies within the top 5% of the lowest *Size* value. Compared to the KLFA benchmarks, models from this region will violate, on average, fewer constraints.

Answer RQ3 (Fault revealing): In mutation testing, the quality of a test suite is evaluated according to its ability to detect known bugs. In this paper we evaluate the quality of our inferred models by checking how many known bug traces a model accepts. The more bug traces a model accepts the better, because we can generate more fault revealing test cases from the model.

We collected 37 bug traces for Tudu and nine bug traces for Cyclos. For each model generated by KLFA, multi-objective GA and NSGA-II, we then constructed all event sequences up to length $L = 6, L = 8$ and $L = 10$ in a depth-first manner. If a bug trace is a sub-sequence in any of the event sequences, we consider the bug as revealed.

Further, we estimate the *effort* for generating a bug trace as the ratio between number of event sequences to be generated (up to length L) and number of bugs revealed. Table VI summarizes the average number of traces and, on average, how many bugs we are able to detect with those traces.

1) Tudu-LL: Using a maximum sequence length $L = 6$, we can use the KLFA model to detect 14 Tudu bugs. This compares to an average of three bugs found by the multi-objective GA and NSGA-II models. However, the NSGA-II inferred models have the best event sequence to bug ratio.

If we increase the maximum sequence length L to eight, we encountered "OutOfMemory" exceptions for the KLFA model, despite increasing the Java stack size to 6GB. The number of event sequences we can generate given $L = 8$ is simply too big.

Increasing L to eight does not change the number of bugs found for models inferred by the multi-objective algorithms. If we increase L to ten however, we can detect an average of four bugs using the multi-objective GA models. Models inferred from the NSGA-II can still only detect an average of three bugs.

2) Tudu-HL: Given $L = 6$, we can detect 25 bugs using the KLFA model. Models generated by the NSGA-II detect on average three bugs. However, models inferred from the multi-objective GA cannot be used to detect any bug. Again, in terms of effort to find a bug, the NSGA-II inferred models have the

best event sequence to bug ratio.

As before we cannot increase L beyond six for the KLFA model. Increasing L to eight makes no difference for the multi-objective GA or NSGA-II models. Once we increase L to ten, the multi-objective GA models detect on average one bug, while the the NSGA-II models detect on average four bugs. However, as L increases, the bug detection ratio decreases.

3) Cyclos: With $L = 6$, we can generate 328 event sequences from the KLFA model, on average 14 event sequences from the multi-objective GA inferred models, and on average 52 event sequences from the NSGA-II inferred models. However, none of these event sequences matches a bug trace. Increasing L to eight or ten makes no difference either. This means `Cyclos` bugs are hard to detect, requiring potentially many event sequences (of length $L > 10$) before a bug trace can be generated.

B. Discussion

In this section we examined three properties of models inferred by the multi-objective GA, NSGA-II and KLFA. We found that KLFA models over-approximate an application's behaviour. A side-effect of over-approximation is that KLFA models violate more application constraints than models found by multi-objective optimization. On the other hand, over-approximating models are better suited to find bugs. For example, in theory, the KLFA model for `Cyclos` could detect nine bugs, the model for `Tudu-LL` 14 bugs and the model for `Tudu-HL` 25 bugs. These numbers were computed by checking if the KLFA model contains a state that accepts a bug trace.

There is however a cost involved in finding bugs. Due to the large number of event sequences that can be generated from KLFA models (given a maximum sequences length L), the ratio of event sequences to bugs detected is very low. In the case of `Cyclos` for example, it was impossible to generate any bug trace from the KLFA model in practice. Despite detecting fewer bugs overall, the multi-objective algorithms, in particular the NSGA-II, infer models that have a better event sequence to bug detection ratio.

In summary, multi-objective optimization results in models that are well-distributed across several combinations of the levels of over- and under-approximation. The tester can choose a region from these trade-offs to ensure that few constraints are violated. This results in fewer infeasible test cases. These models also have a very favourable ratio of event sequences generated per bug revealed, although they cannot reveal all bugs in the application under test.

V. Related Work

A vast literature exists on the inference of finite state models from a set of observed execution traces [1], [4], [5], [2], [11], [13]. The algorithms used for inference perform either an abstraction of the observed concrete states [5] or an abstraction of the observed event sequences [2]. In both cases, the result of the abstraction is a finite state machine which accepts more traces than the observed ones and might (in some cases) not accept some of the observed traces. The capability to accept more traces than just the observed traces is a desired property of the inferred model, which is supposed to generalize the observed behaviours into the set of all possible behaviours.

Existing algorithms are sometimes evaluated for their generalization capabilities, using metrics such as the balanced classification rate [23], which takes into account both false positives (associated with over-approximation) and false negatives (associated with under-approximation).

However, to the best of our knowledge, no approach has ever incorporated the two (implicit) objectives of minimizing both over- and under-generalization. This work is the first to explicitly address the problem of over- and under-generalization by means of a multi-objective optimization approach, and using positive examples only (*i.e.*, only legal traces are used for the inference).

The foundation of model inference from execution traces lies in the theory of grammar inference and more specifically automata induction [3], [19]. Most applications of automata induction to software engineering [2], [13] extend and adapt grammar inference algorithms to the problem of generating a model from traces produced during the execution of a software system. The inferred model is expected to be useful for carrying out some software engineering task, typically model based testing [13], but also formal verification [1] and intrusion detection [18]. There are two major families of algorithms that perform grammar inference from event sequences: (1) algorithms relying on positive examples only [2], [4], [7], [13], [21]; (2) algorithms that assume the availability of both positive and negative examples [3], [9], [12], [20].

Our work clearly belongs to the first category, since negative examples are usually not available when software traces are considered. Some approaches [12], based on the availability of both positive and negative examples, formulate the model inference problem as an optimization problem. In fact, they measure precision and recall in order to maximize their harmonic mean (also known as F-measure).

However, these approaches are single-objective, while ours is multi-objective. However, the main difference with our approach is that we work under the assumption that negative examples are not available, which makes precision and recall not computable in our case.

Some approaches [1] aim at inferring a model of the *common* behaviour of the software, under the assumption that a common behaviour is often also a correct behaviour, that can be used as the basis for the definition of a formal specification of the system. The algorithm used for this purpose is a probabilistic finite state automaton learner, optimized in terms of size (smaller / better) and capability of recognizing frequent sub-sequences (low *UnRec* for frequent sub-traces). Differently from our approach, the over-generalization problem is not addressed directly and explicitly, being (to some extent) a side-effect of the probabilistic automata induction algorithm.

TABLE VI

THE 'AVG. # TRACES' COLUMN SHOWS THE AVERAGE NUMBER OF TRACES (UP TO LENGTH L) WE ARE ABLE TO GENERATE FROM ALL MODELS ON A PARETO FRONT OR THE KLFA MODEL. THE 'AVG. # BUGS' COLUMN SHOWS THE AVERAGE NUMBER OF BUGS THESE TRACES DETECT.

Subject	Algorithm	($L = 6$)		($L = 8$)		($L = 10$)	
		Avg. # Traces	Avg. # Bugs	Avg. # Traces	Avg. # Bugs	Avg. # Traces	Avg. # Bugs
`Tudu LL`	GA	15	3	29	3	101	4
`Tudu LL`	NSGA-II	8	3	9	3	13	3
`Tudu LL`	KLFA	4224	14	–	–	–	–
`Tudu HL`	GA	81	0	1103	0	16967	1
`Tudu HL`	NSGA-II	41	3	76	3	177	4
`Tudu HL`	KLFA	928232	25	–	–	–	–
`Cyclos`	GA	14	0	45	0	178	0
`Cyclos`	NSGA-II	52	0	128	0	276	0
`Cyclos`	KLFA	328	0	–	–	–	–

VI. CONCLUSIONS AND FUTURE WORK

This paper has investigated the use of a multi-objective Genetic Algorithm and the NSGA-II to infer models from execution traces for testing. Inferred models balance the amount of over- and under-approximation of an application's behaviour.

We found models generated by multi-objective algorithms to be well-distributed across various levels of over- and under-approximation. In terms of validity, the multi-objective algorithms produce models that violate fewer application constraints than the KLFA models. On the other hand, models produced by KLFA could potentially reveal more bugs. However, the ratio of the number of event sequences we have to generate from KLFA models in order to find a bug is prohibitively high. The number of event sequences (hence, test cases) that need to be generated from multi-objective models per bug revealed is substantially smaller, making the latter models preferable in practice.

In our future work we plan to further assess the generality of the inferred models, by conducting new experiments. We also plan to experiment with machine learning approaches based on the use of both, a training and a validation set, during model inference.

ACKNOWLEDGEMENT

Mark Harman, Yue Jia, Kiran Lakhotia, Alessandro Marchetto, Cu Duy Nguyen and Paolo Tonella are funded through the EU project FITTEST (ICT-2009.1.2 no 257574). In addition, Mark Harman is supported by EPSRC Grants EP/G060525/1, EP/D050863, GR/S93684 & GR/T22872 and also by Daimler Berlin, BMS and Vizuri Ltd. London.

REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 4–16.

[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] O. Cicchello and S. C. Kremer, "Inducing grammars from sparse data sets: A survey of algorithms and results," *Journal of Machine Learning Research*, vol. 4, pp. 603–632, 2003.

[4] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 215–249, 1998.

[5] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *Proceedings of the 2006 international workshop on Dynamic systems analysis*, ser. WODA '06. New York, NY, USA: ACM, 2006, pp. 17–24.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE-EC*, vol. 6, pp. 182–197, Apr. 2002.

[7] F. Denis, A. Lemay, and A. Terlutte, "Learning regular languages using rfsas," *Theor. Comput. Sci.*, vol. 313, no. 2, pp. 267–294, 2004.

[8] J. Dubois, "Tudu Lists," http://www.julien-dubois.com/tudu-lists.

[9] M. J. Heule and S. Verwer, "Exact dfa identification using sat solvers," in *Grammatical Inference: Theoretical Results and Applications 10th International Colloquium, ICGI 2010*, ser. Lecture Notes in Computer Science, J. M. Sempere and P. Garca, Eds., vol. 6339. Springer, 2010, pp. 66–79.

[10] J. H. Holland, *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[11] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic, "Using dynamic execution traces and program invariants to enhance behavioral model inference," in *ICSE (2)*, 2010, pp. 179–182.

[12] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish, "Regular expression learning for information extraction," in *Conference on Empirical Methods in Natural Language Processing*, 2008, pp. 21–30.

[13] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *30th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2008.

[14] A. Marchetto and P. Tonella, "Search-based testing of Ajax Web applications," in *1st International Symposium on Search Based Software Engineering*, May 2009, pp. 3 –12.

[15] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax Web applications," in *ICST*, 2008, pp. 121–130.

[16] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *ISSRE*. IEEE Computer Society, 2008, pp. 117–126.

[17] L. Mariani and M. Pezzè, "Dynamic detection of cots component incompatibility," *IEEE Software*, vol. 24, no. 5, pp. 76–85, 2007.

[18] C. C. Michael and A. K. Ghosh, "Using finite automata to mine execution data for intrusion detection: A preliminary report," in *Recent Advances in Intrusion Detection*, 2000, pp. 66–79.

[19] R. Parekh and V. Honavar, "Grammar inference, automata induction, and language acquisition," in *Handbook of Natural Language Processing*. Marcel Dekker, 2000, pp. 727–764.

[20] S. Porat and J. A. Feldman, "Learning automata from ordered examples," *Machine Learning*, vol. 7, pp. 109–138, 1991.

[21] S. P. Reiss and M. Renieris, "Encoding program executions," in *23rd International Conference on Software Engineering (ICSE)*, 2001, pp. 221–230.

[22] STRO Uruguay and Instrodi, "Cyclos," http://project.cyclos.org.

[23] N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, and P. Dupont, "A framework for the competitive evaluation of model inference techniques," in *Proceedings of the International Workshop on Model Inference in Testing (MIIT)*, 2010.