Structural Statistical Software Testing with Active Learning in a Graph

Nicolas Baskiotis, Michèle Sebag

Structural Statistical Software Testing with Active Learning in a Graph

Nicolas Baskiotis[1] and Michele Sebag[1]

CNRS − INRIA − Université Paris-Sud
LRI Bat 490, F-91405 Orsay, France
Nicolas.Baskiotis, Michele.Sebag @lri.fr

Abstract. Structural Statistical Software Testing (SSST) exploits the control flow graph of the program being tested to construct test cases. Specifically, SSST exploits the *feasible paths* in the control flow graph, that is, paths which are actually exerted for some values of the program input; the limitation is that feasible paths are massively outnumbered by infeasible ones. Addressing this limitation, this paper presents an active learning algorithm aimed at sampling the feasible paths in the control flow graph. The difficulty comes from both the few feasible paths initially available and the nature of the feasible path concept, reflecting the long-range dependencies among the nodes of the control flow graph. The proposed approach is based on a frugal representation inspired from Parikh maps, and on the identification of the conjunctive subconcepts in the feasible path concept within a Disjunctive Version Space framework. Experimental validation on real-world and artificial problems demonstrates significant improvements compared to the state of the art.

Key words: Structured Sampling, Structured Active Learning, Structural Statistical Software Testing, Disjunctive Version Space, Machine Learning Application to Computer Science

1 Introduction

Autonomic Computing is becoming a new application domain for Machine Learning (ML), motivated by the increasing complexity of current systems [1]. Ideally, systems should be able to automatically adapt, maintain and repair themselves; a first step to this end is to build self-aware systems, using ML to automatically model the system behavior. Similar trends are observed in the field of software design; various ML approaches have been proposed for Software Testing [2, 3], Software Modeling [4] and Software Debugging [5].

Resuming an earlier work [3], this paper is motivated by Statistical Structural Software Testing (SSST) [6]. SSST exploits the control flow graph of the program being tested (Fig. 1) to construct test cases; specifically, test cases are derived from the *feasible paths* in the control flow graph, that is, paths which are actually exerted for some values of the program input. However, for reasonable

size programs there is a huge gap between the syntactical description of the program (the control flow graph) and its semantics (the feasible paths). In practice, the fraction of feasible paths might be as low as 10^{-5} for small size programs, making it inefficient to uniformly sample the paths in the control flow graph.

The characterization of the feasible path region faces several difficulties. First of all, the target concept (i.e. the feasible path region) is non-Markovian: a path is infeasible as it violates some subtle, long-range dependencies among the program nodes. A frugal propositional representation extending Parikh maps [7] was proposed in [3], allowing one to express node dependencies in a compact way. However, using either a relational or a propositional representation, supervised learning was found to fail; this failure was blamed on the very few feasible paths initially available, due to their high computational cost.

Meanwhile, SSST is primarily interested in acquiring more feasible paths, suggesting that an active learning approach [8] might be more relevant than a supervised learning one. In [3], a probabilistic generate-and-test approach called *EXIST* (*Exploration−Exploitation Inference for Software Testing*), built on the top of the extended Parikh map representation was proposed to generate new feasible paths. The limitation of this approach is due to the highly disjunctive nature of the target concept, blurring the probability estimates. In the current paper, the latter limitation is addressed using a bottom-up algorithm inspired from the Version Space [9], called *MLST* for *ML-based Sampling for Statistical Structural Software Testing*, which identifies the conjunctive subconcepts in the target concept. Empirical validation on real-world and artificial problems shows that *MLST* significantly improves on the state of the art.

The paper is organized as follows. Section 2 introduces the formal background and prior knowledge related to the SSST problem; it discusses the limitations of supervised learning for SSST and describes the extended Parikh representation. Section 3 gives an overview of the *MLST* algorithm. Section 4 reports on the empirical validation of *MLST* on real-world and artificial problems, and discusses the approach compared to the state of the art. The paper concludes with some perspectives for further research.

2 Position of the problem

This section introduces statistical structural software testing (SSST), situates the problem in terms of supervised learning, and describes the extended Parikh map representation used in the following.

2.1 Statistical Structural Software Testing

Many Software Testing methods are based on the generation of test cases, where a test case associates a value to every input variable of the program being tested. For each test case, the program output is compared to the expected output (e.g., determined after the program specifications) to find out misbehaviors or bugs in the program implementation. The test quality thus reflects the coverage of

the test cases (see below). Statistical testing methods, enabling intensive test campaigns, most often proceed by sampling the input space. However, uniform sampling is bound to miss the exception branches (e.g. calling the division routine with denominator = 0), the measure of which is null. More generally, uniformly sampling the input domain does not result in a good coverage of the execution paths of the program. In order to overcome this limitation, a method combining statistical testing and structural analysis, based on the control flow graph of the program being tested (Fig. 1) was proposed by [6].

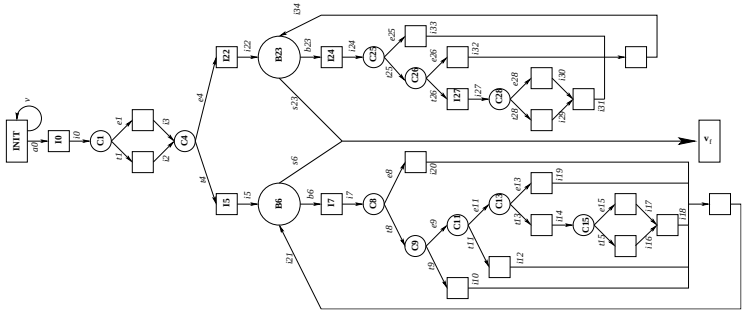


Fig. 1. Program FCT4 includes 36 nodes and 46 edges.

The control flow graph provides a syntactical representation of the program. Formally, the control flow graph is a Finite State Automaton (FSA) noted $(\Sigma, \mathcal{V})$ where:

• Σ is the set of program nodes, a node being either a condition or a block of instructions, and

• $\mathcal{V}$ specifies the allowed transitions between the nodes.

For every node v in Σ, $Suc(v)$ denotes the set of successors of v, i.e. the set of all nodes w such that transition (v, w) belongs to $\mathcal{V}$. A program path noted s is represented as a finite length string on Σ, obtained by iteratively choosing a node among the successors of the current node until reaching the final node v_f.

The semantics of the program is expressed by the fact that not every path in the FSA is *feasible*, i.e. is such that the path is actually executed for some values of the program input variables. The infeasibility of a given path arises as it violates some dependencies between different parts of the program or it does not comply with the program specifications. Two most general causes for path infeasibility are the **XOR** and the **Loop** patterns.

XOR pattern. Given a program where two `if` nodes are based on some (unchanged) expression, the successors of these nodes will be correlated in every feasible path: if the successor of the first `if` node is the `then` (respectively, `else`) node, then the successor of the second `if` node must be the `then` (resp. `else`) node. Such patterns, referred to as *XOR* patterns, express the possibly long-range dependencies between the fragments of the program paths.

Loop pattern. The number of times a loop is executed happens to be restricted by the semantics of the application; e.g. when the problem involves 18 or 19 uranium beams to be controlled, the control procedure will be executed exactly 18 or 19 times [10]. This pattern is referred to as *Loop* pattern.

An upper bound T on the length of the considered paths is set by the software testing expert for practical reasons, although path length is usually unbounded (since programs generally involve `repeat` and `while` instructions). Thanks to this upper bound, one can use well-known results from labelled combinatorial structures [11] to uniformly sample the T-length paths in the control flow graph [6]. Eventually, every path is rewritten as a Constraint Satisfaction Problem, expressing the set of conditions on the input variables of the program ensuring that the path is exerted. If the Constraint Solver (CS) finds a solution, the path is labelled *feasible* and the solution precisely is the test case; otherwise the path is *infeasible*.

As already mentioned, the main limitation of this approach is when the fraction of feasible paths is tiny, which is the general case for medium length programs [6]. In such cases, the number of retrieved test cases remains insufficient while the computational effort dramatically increases; it needs some days of computation to find out a few dozen or hundred test cases. The software testing expert then inspects the program, manually decomposing the control flow graph and/or adding conditions in order to make it easier to find feasible paths.

2.2 SSST and Supervised Learning

In order to support Statistical Structural Software Testing, one possibility is to use supervised Machine Learning, exploiting a sample of labelled paths as training set. From such a training set $\mathcal{E} = \{(s_i, y_i), s_i \in \Sigma^T, y_i \in \{-1, +1\}, i = 1 \ldots, n\}$, where s_i is a path with length at most T and y_i is 1 iff s_i is feasible, supervised ML can be made to approximate the program semantics, specifically to construct a classifier predicting whether some further path is *feasible* or *infeasible*. Such a classifier would be used as a pre-processor filtering out the paths that are deemed infeasible and thus significantly reducing the CS computational cost.

In a supervised learning perspective, the SSST application presents some specificities. Firstly, it does not involve noise, i.e. the oracle (constraint solver) does not make errors[1]. Secondly, the complexity of the example space is huge with respect to the number of available examples. In most real-world problems, Σ includes a few dozen symbols; a few dozen paths are available, each a few hundred symbols long. The number of available paths is limited by the labelling cost, i.e. the runtime of the constraint solver (on average a few seconds per program path). Thirdly, the data distribution is severely imbalanced (infeasible paths outnumber the feasible ones by several orders of magnitude). Lastly, the

[1] Three classes should be considered (feasible, infeasible and undecidable) as in all generality the CSPs are undecidable. However the undecidable class depends on the constraint solver and its support is negligible in practice.

label of a path depends on its global structure; many more examples would be required to identify the desired long-range dependencies between the transitions, within a Markovian framework. Specifically, probabilistic FSAs and likewise simple Markov models can hardly model the infeasibility patterns such as the XOR or Loop patterns. While Variable Order Markov Models [12] could accommodate such patterns, they are ill-suited to the sparsity of the initial data available.

In summary, supervised learning is impaired by the poor quality of the available datasets relatively to the complexity of the instance space.

2.3 Extended Parikh representation

A frugal and flexible representation inspired by Parikh maps was proposed in [3] in order to characterize conjunctions of XOR and Loop patterns in a compact way. Parikh maps [7, 13] characterize a string from its histogram with respect to alphabet Σ; to each symbol v in Σ is associated an integer attribute $|\cdot|_v$ defined on Σ^*, where $|s|_v$ is the number of occurrences of v in string s.

As this representation is clearly insufficient to account for long range dependencies in the strings, additional attributes are defined. To each pair (v, i) in $\Sigma \times \mathbb{N}$ is associated an attribute $|\cdot|_{v,i}$, from Σ^* onto Σ, where $|s|_{v,i}$ is the successor of the i-th occurrence of the v symbol in s, or v_f if the number of v occurrences in s is less than i. Extended Parikh maps have a low representation

$$\begin{array}{rlll} v \in \Sigma & |\cdot|_v : \Sigma^* \mapsto \mathbb{N} & |s|_v = \#v \ in \ s \\ (v,i) \in \Sigma \times \mathbb{N} & |\cdot|_{v,i} : \Sigma^* \mapsto \Sigma & |s|_{v,i} = \text{ successor of i-th occurrence of } v \end{array}$$

$s = vwvtxytx \quad \rightarrow$

	$\lvert\cdot\rvert_v$	$\lvert\cdot\rvert_w$	$\lvert\cdot\rvert_t$	...	$\lvert\cdot\rvert_{v,1}$	$\lvert\cdot\rvert_{v,2}$	$\lvert\cdot\rvert_{w,1}$	$\lvert\cdot\rvert_{t,1}$
s	2	1	2		w	t	v	x

Table 1. Extended Parikh representation. An example.

complexity; the number of propositional attributes is $|\Sigma| \times k$ where $k << T$ is the maximal number of occurrences of any symbol in a T-length string.

However, supervised learning within extended Parikh maps fails too. A partial conclusion is that supervised ML requires more feasible paths than normally available in SSST problems. Meanwhile, SSST is also primarily interested in building feasible paths. A new learning goal is thus defined.

3 Overview

This section describes the *MLST* system aimed at the generation of new feasible paths based on the initial training set $\mathcal{E}$. "Feasible path" and "positive example" are interchangeably used in the remainder of the paper.

3.1 Principle

Every new path s is constructed iteratively; s is initialized to the start symbol (the root node of the control flow graph); at each time step a new symbol is selected and concatenated to s. Let v be the current last symbol in s, and denote $s.w$ the concatenation of s and w. In each time step one should select the symbol w such that $s.w$ is the prefix of many feasible paths; formally:

$$\begin{aligned} Select\ w^* &= argmax\{p_w,\ w \in Suc(v)\} \\ p_w &= Pr(s'\ feasible\ |Prefix(s') = s.w) \end{aligned} \tag{1}$$

However, the above criterion suffers from two limitations. Firstly, the set of strings s' with prefix $s.w$ is almost always empty after the first iterations (due to the size of the training set $\mathcal{E}$); and in the first iterations, this criterion would lead to duplicate the known feasible paths whereas the goal is to find *new* feasible paths. This first limitation was addressed as i) the conditioning on $Prefix(s') = s.w$ was replaced by a generalization thereof, and ii) an ϵ-greedy selection was used (section 3.3).
The second limitation comes from the fact that, after prior knowledge (section 2.1) the feasible path concept involves the conjunction of quite a few XOR patterns. With respect to the extended Parikh map representation, the target concept tc thus involves the disjunction of many conjunctive subconcepts:

$$tc = C_1 \vee \ldots \vee C_K$$

When several s' belonging to different C_i are used to estimate p_w, this estimate can be misleading; mixing the evidence derived from paths belonging to different C_i does not provide reliable indications, for the same reason as selecting the attribute with maximal entropy in a decision tree usually is inappropriate when learning a disjunctive concept. In the *EXIST* algorithm [3], this limitation was partly addressed by using the *Seeding* heuristics, stochastically extracting subsets of positive examples such that their least general generalization does not cover any negative example, referred to as admissible subsets; in each step, conditional probabilities p_w are computed from a single admissible subset. The rationale for the *Seeding* heuristics is that an admissible subset should mostly contain positive examples belonging to the same subconcept C_i; in practice, this heuristics was found to significantly improve the *EXIST* performances [3]. However, the *Seeding* heuristics suffers from two limitations. On the one hand, the initial negative examples are insufficient and do not prevent the admissible subsets from spanning over several subconcepts C_i, thus corrupting the p_w; on the other hand, the *Seeding* heuristics tends to oversample the subconcepts C_i which are best represented in the training set.
The *MLST* algorithm addresses both above limitations through a principled characterization of all subconcepts C_i represented in the training set, through the Init module (section 3.2).

Finally, *MLST* is organized as follows. The Init module (section 3.2) aims at a maximally specific disjunctive description of the initial feasible paths (the

S set, in terms of Disjunctive Version Space); it constructs conjunctive subconcepts $\hat{C}_1, \ldots \hat{C}_J$, where with high probability each $\hat{C}_i$ is a specialization of some C_j represented in the training set[2]. The Generalization module (section 3.4) independently generalizes each $\hat{C}_i$. Both modules rely on the Constrained Exploration module (section 3.3). Both Init and Generalization modules interact with the Oracle (the constraint solver), labelling every newly generated path as *feasible* or *infeasible*.

3.2 Init Module

The Init module is inspired from the Version Space framework [9]. Let the binary predicate $\mathcal{R}(s, s')$ be defined as true iff both s and s' belong to some conjunctive subconcept C_i. The Init module thus computes a stochastic estimate of $\mathcal{R}(s, s')$ noted $\hat{\mathcal{R}}(s, s')$, and uses it to construct cliques of the positive examples. With high probability (depending on the accuracy of $\hat{\mathcal{R}}$, see below), all examples in such a clique belong to the same C_i; therefore their least general generalization (lgg) defines a specialization of C_i, noted $\hat{C}_i$.

The construction of relation $\hat{\mathcal{R}}$ proceeds as follows. By definition, $\mathcal{R}(s, s')$ holds iff $lgg(s, s')$ is correct, i.e. does not cover any infeasible path. Prior knowledge on the problem domain suggests that the target concept has a tiny and fragmented coverage (section 2.1); therefore, if $\mathcal{R}(s, s')$ does not hold, then any path generated in $lgg(s, s')$ will be infeasible with high probability. Accordingly, a stochastic approximation of $\mathcal{R}(s, s')$ is implemented (Fig. 3.2), calling the Constrained Exploration module to independently generate and label p paths in $lgg(s, s')$. If all p paths are feasible, $\hat{\mathcal{R}}(s, s')$ returns true, otherwise it returns false and the infeasible paths are added to the set $\mathcal{E}^-$ of infeasible paths. Clearly $\hat{\mathcal{R}}(s, s')$ implements a complete but incorrect approximation of $\mathcal{R}(s, s')$; its accuracy (1 - $\Pr(\hat{\mathcal{R}}(s, s')|\neg\mathcal{R}(s, s'))$) goes to 0 exponentially with p; a typical value for p in the experiments (section 4) is $p = 2$.

After $\hat{\mathcal{R}}(s, s')$ has been computed for all pairs of training feasible paths, the maximal clique $\widehat{C}(s)$ covering each feasible training path s (not already covered) is computed using a standard greedy algorithm (Fig. 3). At the j-th step, $\mathcal{V}_j$ includes all examples related by $\hat{\mathcal{R}}$ to all elements in S_j (with $S_0 = \{s\}$). If $\mathcal{V}_j$ is empty, stop; otherwise, the example in $\mathcal{V}_j$ related to most examples in $\mathcal{V}_j$ by $\hat{\mathcal{R}}$ is selected, added to S_j and removed from $\mathcal{V}_j$. Finally, the Init module produces a set of cliques noted $\hat{C}_i$. It is straightforward to show that with high probability, for each C_k represented in the training set there will be some $\hat{C}_i$ such that $\hat{C}_i$ is a specialization of C_k; the probability exponentially increases with the number of training examples in C_k and parameter p used to compute $\hat{\mathcal{R}}$.

By abuse of notations, any clique $\hat{C}$ is viewed as both a set of feasible paths and their lgg.

[2] The identification of conjunctive subconcepts not represented in the training set is left for further study.

Input: set $\mathcal{E}^-$ of infeasible paths.
Parameter $p \in \mathbb{N}$
For all $s'' \in \mathcal{E}^-$
If s'' is covered by $lgg(s, s')$
return False
For $i = 1$ to p
s''= C.Exploration $(lgg(s, s'))$
If (label(s'') = infeasible)
Add s'' to $\mathcal{E}^-$
Return False
Return True

Fig. 2. Routine $\hat{\mathcal{R}}(s, s')$

$S_0 = \{s\}$
$j = 1$
$\mathcal{V}_j = \{s'/\hat{\mathcal{R}}(s', s'') for\ all\ s'' \in S_{j-1}\}$
$Degree_j(s') = |\{s''\ in\ \hat{\mathcal{V}}_j/\mathcal{R}(s', s'')\}|$
While $\mathcal{V}_j$ is not empty
$s' = argmax_{\mathcal{V}_j}\{Degree(s'')\}$
$S_j = S_{j-1} \cup \{s'\}$
$\mathcal{V}_{j+1} = \mathcal{V}_j \setminus \{s'\}$
Increment j
EndWhile
Return S_j

Fig. 3. Routine $\widehat{C}(s)$

3.3 Constrained Exploration module

Given a constraint h on paths, the Constrained Exploration module aims to generate a path s such that $h(s)$ holds. This module is called by the Init module using a syntactic constraint ($h(s) \equiv s$ belongs to $lgg(s', s'')$) and in the Generalization module using a semantic constraint ($h(s) \equiv s$ is feasible).

Along the same lines as in section 3.1, the Constrained Exploration module proceeds iteratively, initializing path s to the starting symbol and selecting in each time step the successor of the last symbol in s noted v, in order to maximize the probability for s to ultimately satisfy h. While one might want to select w maximizing the frequency of $h(s')$ over all strings s' with prefix $s.w$, (eq. 1), in most cases there is no such s'. The conditioning on $Prefix(s') = s.w$ is thus relaxed. Let v and i respectively denote the last symbol in s, and its number of occurrences ($|s|_v = i$). Condition $Prefix(s') = s.w$ is generalized as: s' is such that the successor of the i-th occurrence of the v symbol is w and the number of occurrences of w in s' is strictly greater than in s $(|s'|_{v,i} = w) AND (|s'|_w > |s|_w)$. If such paths s' exist among the available ones (training examples and examples generated along the process), frequency q_w is defined as:

$$q_w = Pr(h(s') \mid ((|s'|_{v,i} = w) AND (|s'|_w > |s|_w))) \tag{2}$$

Note that, as only standard programs are considered, any symbol (program node) has at most two successors. Letting w and w' denote the two successors of the last node v, the node selection routine thus considers three cases:

- If both q_w and $q_{w'}$ are defined, the node with maximal frequency is selected (select $argmax\{q_w, w \in Suc(v)\}$;
- If neither q_w nor $q_{w'}$ is defined, one node is selected randomly;
- Otherwise (say that q_w is defined and $q_{w'}$ is not):
  - in the Init framework, w is selected;
  - in the Generalization framework, an ϵ-greedy selection is used: w is selected with probability $1 - \epsilon$ and w' is selected with probability ϵ.

Obviously, this procedure does not guarantee that path s will satisfy $h(s)$; however, as every newly generated path is added to the available examples,

and accordingly bias the computation of q_w, a fast convergence toward paths complying with constraint h was empirically observed (see section 4).

3.4 Generalization Module

The Generalization module aims at maximally generalizing every $\hat{C}$ produced by the Init module, by generating new paths s "close" to $\hat{C}$ and adding them to $\hat{C}$ if they are labelled feasible.

This module exploits the Constrained Exploration module with constraint $h(s) \equiv s$ is feasible, with a single difference: q_w is estimated from i) only feasible paths satisfying $\hat{C}$; ii) only infeasible paths generated when generalizing $\hat{C}$. This restriction in the computation of q_w overcomes the limitations discussed in section 3.1, due to the disjunctive nature of the target concept.

Several heuristics have been investigated as alternatives to the ϵ-greedy selection in the Constrained Exploration module; for instance, a more sophisticated Exploration vs Exploitation trade-off based on the multi-armed bandit UCB algorithm [14] was considered; however, UCB-like approaches were penalized as the "reward" probability is very low (being reminded that the fraction of feasible paths commonly is below 10^{-5}).

4 Experimental Validation

This section presents the experimental setting and goals, and reports on the results of *MLST*.

4.1 Experimental Setting

MLST is first validated on the real-world Fct4 program, including 36 nodes and 46 edges (Fig. 1). The ratio of feasible paths is circa 10^{-5} for a maximum path length $T = 250$. Fct4 is a fragment of a program used for a safety check in a nuclear plant [10].

For the sake of extensive validation, a stochastic problem generator was also designed, made of two modules. The first module defines the "program syntax", made of a control flow graph generated from a probabilistic BNF grammar[3]. The second module constructs the "program semantics", or target concept tc, determining whether a path in the above graph is feasible. After prior knowledge (section 2.1), the target concept is a conjunction of XOR patterns and Loop patterns. In order to generate satisfiable target concepts, a set $\mathcal{P}$ of paths uniformly generated from the control flow graph is first constructed; iteratively, i) one selects a XOR concept covering a strict subset of $\mathcal{P}$; ii) paths not covered by the

[3] Three non-terminal nodes were considered (the generic structure *block*, the *if* and the *while* structures), together with two terminal nodes (the *Instruction* and the *Condition* node. The probabilities on the production rules control the length and depth of the control flow graph. Instructions are pruned in such a way that each node has exactly two successor nodes; lastly, each node is associated a distinct label.

XOR concept are removed from $\mathcal{P}$. Finally, the target concept tc is made of the conjunction of the selected XOR concepts and the Loop concepts satisfied by the paths in $\mathcal{P}$. The coverage of each conjunction is measured on an independent set of $100,000$ paths uniformly generated in the extension of tc, using [11].

Ten artificial problems are considered, with coverage ratio ranging in $[10^{-15}, 10^{-3}]$, number of nodes in $[20, 40]$ and path length in $[120, 250]$. Ten runs are launched for each problem, considering independent training sets, made of the set $\mathcal{E}_i$ of the initial 50 feasible paths, plus 50 infeasible paths For each $\hat{C}$ identified by the Init module, the Generalization module is launched 400 times. Set $\mathcal{E}_f$ gathers all feasible paths, the initial ones and the newly generated ones.

For each conjunctive subconcept C of the target concept represented in the training set, the performance of the algorithm is assessed by comparing the initial and final coverage of C, defined as follows. Let C_i (respectively C_f) denote the lgg of all paths in $\mathcal{E}_i \bigcap C$ (resp. $\mathcal{E}_f \bigcap C$). The initial coverage of C, noted $i(C)$, is the fraction of paths in C that belong to C_i; symmetrically the final coverage of C noted $f(C)$ is the fraction of paths in C that belong to C_f. Both $i(C)$ and $f(C)$ are estimated from a uniform sample of 10,000 examples in C, generated after [11].

For a better visualization, the average final coverage is computed using a Gaussian convolution over all subconcepts represented in the training set ($\mathcal{E}_i \bigcap C \neq \emptyset$):

$$f(x) = \frac{\sum_{C \cap \mathcal{E} \neq \emptyset} f(C) exp(-\kappa(x - i(C))^2)}{\sum_{C \cap \mathcal{E} \neq \emptyset} exp(-\kappa(x - i(C))^2)}$$

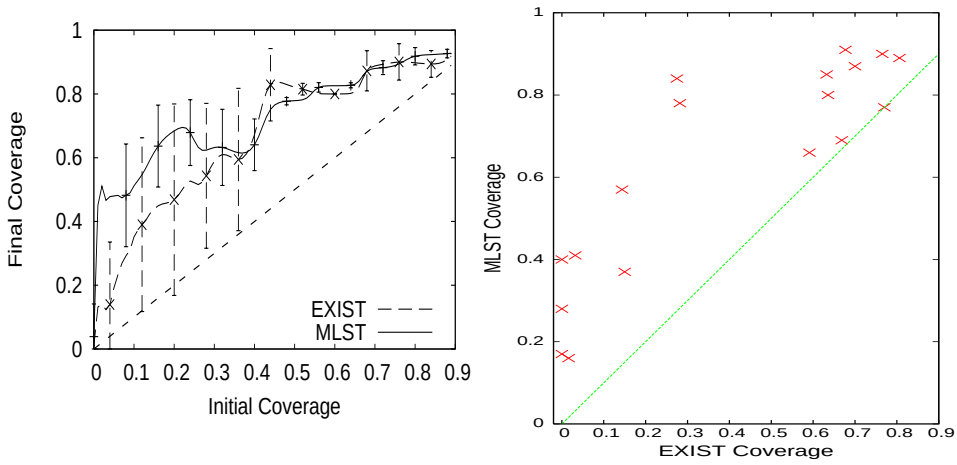
The standard deviation is similarly computed. In both cases, κ is set to 100.

The goal of the experiments is to compare *MLST* with the former *EXIST* algorithm presented in [3] and to assess the added value of the Init module. More precisely, the performance will be examined with respect to the initial coverage of the target subconcepts in the training set.

4.2 Experimental Results

Fig. 4.(a) displays the final vs initial coverage provided by *MLST* on 10 artificial problems, using the ϵ-Greedy generalization module with $\epsilon = .5$, together with the standard deviation; complementary experiments show the good stability of the results for ϵ ranging in $[.1, .8]$. More precise results are presented in Table 2, showing that *MLST* efficiently samples the conjunctive subconcepts that are represented in the training set; when the initial coverage of the subconcept is tiny to small, the gain ranges from 5 to 2 *orders of magnitude*. A factor gain of 3 is observed when the initial coverage is between 10% to 30%.

Fig. 4.(b) reports the gain obtained on the real-world Fct4 problem comparatively to *EXIST* [3] for 10 independent runs with 3000 calls to the constraint solver. The gain of *MLST* is considered excellent by the software testing experts. The computational effort ranges from 3 to 5 minutes (on PC Pentium 3Ghz) for the Init Module and is less than 3 minutes for 400 calls to the generalization module (excluding labelling cost).



(a) Final vs initial coverage $\pm\sigma$ on artificial problems

(b) Comparison with *EXIST* on Fct4

Fig. 4. *MLST* (ϵ=.5): (a) Final vs initial coverage, averaged on 10 artificial problems $\times$ 10 runs; (b) Comparison with *EXIST* on the Fct4 problem (10 runs).

		$[0, 10^{-4}]$	$[10^{-4}, 10^{-3}]$	$[10^{-3}, 10^{-2}]$	$[10^{-2}, 10^{-1}]$	[.1, .3]	[.3, .6]	[.6, 1]
$EXIST$	$log(f/i)$	2.6 ± 2.3	2.6 ± 2.4	$1.1 \pm 1.$	$1.1 \pm 1.$			
	f/i					2.6 ± 1.8	$1.6 \pm .6$	$1.1 \pm .1$
$MLST$	$log(f/i)$	5.7 ± 1.2	5.3 ± 1.2	$3.7 \pm .86$	$2 \pm .72$			
	f/i					$3 \pm .1$	$1.6 \pm .3$	$1.1 \pm .1$

Table 2. *MLST*: Coverage gain averaged on 10 artificial problems $\times$ 10 runs.

Initial coverage	[0, .05]	[.05, .15]	[.2, .4]	[.4, .55]
EXIST final coverage	.01 $\pm$ .01	.1 $\pm$ .06	.44 $\pm$ .16	.71 $\pm$ .05
MLST final coverage	.25 $\pm$ .1	.45 $\pm$ .07	.78 $\pm$ .07	.83 $\pm$ .07

Table 3. Final vs initial coverage of *EXIST* and *MLST* on Fct4 averaged on 10 runs.

As shown in Tables 2 and 3, *MLST* significantly and consistently improves on *EXIST*. In practice, *EXIST* does not much improve the coverage of subconcepts which are poorly represented in the training set; actually, on the FCT4 problem the coverage of small subconcepts ($i < .15$) is left unchanged by *EXIST* whereas *MLST* reaches a coverage of .25 for i in $[0, .05]$ and .45 for i in $[.05, .15]$. This difference is explained as the stochastic *Seeding* heuristics used in *EXIST* must wait for useful negative examples to be generated, in order to construct useful admissible subsets; and in any case, it tends to increase the coverage of subconcepts that are already well represented in the training set. In contrast, *MLST* starts by characterizing all subconcepts C_i which are represented in the training set; thereafter, it spends an equal amount of time on each subconcept, resulting in consistent coverage improvements for all subconcepts, whatever their initial coverage is.

5 Discussion

Few applications of Machine Learning techniques to Software Testing have been proposed in the literature. ML has been used to feed Software Testing with program invariants [15] or characterizing relevant paths in a model checking framework [16]; it has also been used to post-process and generalize the software testing results [2].

A more remotely related work presented by [5] actually focuses on Software Debugging. Indeed, quite a few authors have investigated the generation of test cases for Software Debugging [17–19], most often using Constraint Satisfaction techniques.

The difference between Software Debugging and Software Testing can be characterized in terms of goal as well as quality criterion. For instance when analyzing malware, i.e. malicious software [17], the goal is to detect and prevent fatal errors. In order to do so, one must be able to run every instruction and visit every branch in the program (e.g. finding the test cases triggering malicious instructions). In other words, Software Debugging is interested in test cases enforcing a complete coverage of the block instructions, necessary and sufficient to warrant that the program is not prone to fatal errors.

In Software Testing, the goal is to certify that the software will behave according to its specifications (not every misbehavior causes a fatal error). Therefore, Software Testing aims at a complete coverage of the paths.

Indeed, when the program being tested does not involve loops, there is no difference between the path coverage and the block coverage criteria; in such cases, constraint based approaches are more efficient than ours. Otherwise, obviously the number of paths is infinite (or exponentially larger than the number of block instructions if bounded length paths are considered), and complete path coverage is not tractable. A relaxation of the complete path coverage, the goal of Software Testing thus is to uniformly sample the feasible paths.

Clearly, the distribution of the feasible paths generated by *MLST* is far from being uniform. Further work is concerned with characterizing the distribution of the generated paths, and studying its convergence.

6 Conclusion and Perspectives

The presented application of Machine Learning to Software Testing relies on an efficient representation of paths in a graph, coping with long-range dependencies and data sparsity. Further research aims at a formal characterization of the potentialities and limitations of this extended Parikh representation (see also [20]), in software testing and in other structured domains.

The main contribution of the presented work is to enable the efficient sampling of paths in a graph, targeted at some specific path region. The extension to structured domains of Active Learning, a hot topic in the Machine Learning field for over a decade [21], specifically targeted at the construction of structured examples satisfying a given property, indeed opens new theoretical and applicative perspectives to Relational Machine Learning.

With respect to Statistical Software Testing, the presented approach dramatically improves on former approaches, based on the *EXIST* algorithm [3] and on uniform sampling [6]. Further research is concerned with sampling conjunctive subconcepts which are *not* represented in the initial training set. In the longer run, the extension of this approach to related applications such as equivalence testers or reachability testers for huge automata [22] will be studied.

Acknowledgements

We thank the anonymous reviewers for many helpful comments. The authors gratefully acknowledge the support of `Pascal` Network of Excellence IS T-2002-506 778.

References

1. Rish, I., Das, R., Tesauro, G., Kephart, J.: ECML-PKDD Workshop *Autonomic Computing: A new Challenge for Machine Learning*. (2006)
2. Bréhélin, L., Gascuel, O., Caraux, G.: Hidden Markov models with patterns to learn boolean vector sequences and application to the built-in self-test for integrated circuits. IEEE Transactions Pattern Analysis and Machine Intelligence **23**(9) (2001) 997–1008
3. Baskiotis, N., Sebag, M., Gaudel, M.C., Gouraud, S.D.: Software testing: A machine learning approach. Proceedings of the 20th International Joint Conference on Artificial Intelligence (2007) 2274–2279
4. Xiao, G., Southey, F., Holte, R.C., Wilkinson, D.F.: Software testing by active learning for commercial games. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI). (2005) 898–903

5. Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: simultaneous identification of multiple bugs. In: Proceedings of the 23rd International Conference on Machine Learning. (2006) 1105–1112
6. Denise, A., Gaudel, M.C., Gouraud, S.D.: A generic method for statistical testing. In: Proceedings of the 15th International Symposium on Software Reliability Engineering. (2004) 25–34
7. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
8. Dasgupta, S.: Coarse sample complexity bounds for active learning. In: Advances in Neural Information Processing Systems. (2005) 235–242
9. Mitchell, T.: Generalization as search. Artificial Intelligence **18** (1982) 203–226
10. Gouraud, S.D.: Statistical Software Testing based on Structural Combinatorics. PhD thesis, LRI, Université Paris-Sud (2004)
11. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science **132**(2) (1994) 1–35
12. Begleiter, R., El-Yaniv, R., Yona, G.: On prediction using variable order Markov models. Journal of Artificial Intelligence Research **22** (2004) 385–421
13. Fischer, E., Magniez, F., de Rougemont, M.: Approximate satisfiability and equivalence. In: 21th IEEE Symposium on Logic in Computer Science. (2006) 421–430
14. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning **47**(2-3) (2002) 235–256
15. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27**(2) (2001) 99–123
16. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Actively learning to verify safety for FIFO automata. In: Foundations of Software Technology and Theoretical Computer Science. (2004) 494–505
17. Moser, A., Krügel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: IEEE Symposium on Security and Privacy. (2007) 231–245
18. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: ACM Conference on Computer and Communications Security. (2006) 322–335
19. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (2005) 213–223
20. Clark, A., Florencio, C.C., Watkins, C.: Languages as hyperplanes: Grammatical inference with string kernels. In: Proceedings of the 17th European Conference on Machine Learning. (2006) 90–101
21. Cohn, D.A., Ghahramani, Z., Jordan, M.I.: Active learning with statistical models. In: Advances in Neural Information Processing Systems, The MIT Press (1995) 705–712
22. Yannakakis, M.: Testing, optimization, and games. In: Proceedings of the 31st International Colloquium on Automata, Languages and Programming. (2004) 28–45