Relational Sampling for Statistical Software Testing

Nicolas Baskiotis and Michele Sebag

LRI, Univer. Paris-Sud, CNRS, INRIA
{nbaskiot,sebag}@lri.fr

Abstract. Motivated by Statistical Software Testing, this paper is interested in sampling the feasible paths in a graph, the control flow graph of the program being tested. The difficulty comes from the tiny size of the target region (ranging in $[10^{-10}, 10^{-5}]$ for medium size programs), from the restricted amount of initial examples available and from the non-Markovian nature of the target concept, due to the long-range dependencies between the program nodes.
The new algorithm S4T presented in the paper can be viewed as an Active Relational Learning Algorithm, biased toward sampling new positive examples (feasible paths) and optimizing their diversity.
Experimental validation on real-world and artificial problems demonstrates significant improvements compared to the state of the art.

1 Introduction

Autonomic Computing is becoming a new application domain for Machine Learning (ML), motivated by the increasing complexity of current systems [RDTK06]. 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 behaviour. Along these lines, various ML approaches have been proposed for Software Testing [BGC01], Software Modeling [XSHW05] and Software Debugging [ZJL+06].

Motivated by Statistical Structural Software Testing [DGG04], this paper is concerned with sampling the feasible paths in the control graph of the program being tested. For reasonable size programs, there is a huge gap between the syntactical description of the program (the control flow graph) and its semantics (the set of paths which are actually executed for some configurations of the program input variables, referred to as feasible paths). In practice, the fraction of feasible paths is tiny, ranging in $[10^{-10}, 10^{-5}]$ for medium size programs; this makes the uniform sampling of the control flow graph e.g. based on classical results from labelled combinatorial structures [FZC94] inefficient.

The use of supervised ML in order to characterize the set of feasible paths is severely hindered by the available examples (feasible paths are very expensive and hard to find), on one hand, and by the non-Markovian nature of the underlying target concept on the other hand; a path is infeasible as it violates subtle and usually long-range dependencies among the program nodes. Reinforcement learning (finding a good policy, i.e. walking in the control graph in order to ultimately construct a feasible path) is not applicable as the goal is to find *new* feasible paths.

Using frugal propositional representations inspired from Parikh maps [HU79], propositional learning can be applied to learn an approximation of the "Feasible Path" concept [BSGG07]. However, this characterization is not constructive, i.e. it does not directly allow for generating new feasible paths, which is the core task for Statistical Structural Software Testing. A Generate-and-Test approach built on the top of the Parikh map representation was thus proposed in [BSGG07] to generate new feasible paths.

This paper presents a new algorithm called *S4T* (for *Structural Sampling for Statistical Software Testing*) aimed at sampling the feasible paths. The contribution of *S4T* is to hybridize a probabilistic approach with a divide and conquer heuristics based on the Version Space [Mit82]. Empirical validation on real-world and artificial problems shows that *S4T* significantly improves on the state of the art.

The paper is organized as follows. Section 2 briefly reviews some work relevant to Machine Learning and Software Testing. Section 3 introduces the formal background and prior knowledge related to the SST problem; it discusses the limitations of supervised learning for SST and describes the extended Parikh representation first presented in [BSGG07]. Section 4 gives an overview of the relational active learning *S4T* algorithm. Section 5 reports on the empirical validation of the approach on real-world and artificial problems, and discusses the results compared to the state of the art. The paper concludes with some perspectives for further research.

2 Related Work

Interestingly, while Program Synthesis is among the grand goals of Machine Learning, the application of Machine Learning to Software Testing (ST) has seldom been considered in the literature.

Ernst et al. [ECGN99] aim at detecting program invariants, through instrumenting the program at hand and searching for predetermined regularities (e.g. value ranges) in the traces.

Brehelin et al. [BGC01] consider a deterministic test procedure, generating sequences of inputs for a PLA device. An HMM is trained from these sequences and further used to generate new sequences, increasing the test coverage.

In [VSVA04], the goal is to test a concurrent asynchronous program against user-supplied constraints (model checking). Grammatical Inference is used to characterise the paths relevant to the constraint checking.

Xiao et al. [XSHW05] aim at testing a game player, e.g. discovering the regions where the game is too easy/too difficult; they use active learning and rule learning to construct a model of the program. A more remotely related work presented by [$\mathrm{ZJL}^{+}06$], is actually concerned with software debugging and the identification of trace predicates related to the program misbehaviours.

In [ECGN99,VSVA04], ML is used to provide better input to ST approaches; in [BGC01], ML is used as a post-processor of ST. In [XSHW05], ML directly provides a model of the black box program at hand; the test is done by manually inspecting this model.

3 Position of the problem

This section introduces statistical software testing (SST) and discusses how Machine Learning can be made to support SST. The representation used throughout the paper, based on extended Parikh maps, is last described.

3.1 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 (determined e.g. after the program specifications) to find out misbehaviours or bugs in the program implementation. The quality of the test 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; the drawback is that rare cases, e.g. exceptions, are difficult to retrieve without structural analysis. In order to overcome this limitation, [DGG04] introduce a method combining statistical testing and structural analysis, based on the control flow graph of the program being tested (Fig. 1).

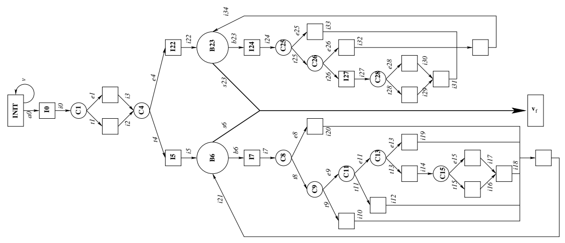


Fig. 1. Program FCT4 includes 36 nodes and 46 edges.

The control flow graph provides a syntactical representation of the program. Formally, the control flow graph is a Finite State Automaton (FSA) based on some finite alphabet Σ, where Σ includes the program nodes (conditions, blocks of instructions), and the FSA specifies the transitions between the nodes. A program path is represented as a finite length string on Σ, obtained by iteratively choosing a node among the successors of the current node until the final node noted v_f is found.

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(n) 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. This pattern is referred to as *Loop(n)* pattern.

While the length of program paths is not upper bounded in general, for practical reasons coverage-based approaches to software testing consider program paths with bounded length T. Well-known results from labelled combinatorial structures [FZC94] thus enable the uniform sampling of the T-length paths in the control flow graph [DGG04]. 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 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 [DGG04]. In such cases, the number of retrieved test cases remains insufficient while the computational effort of the CSP resolution increases dramatically; it needs some days of computation to find out a few dozen or hundred test cases. The test expert then proceeds by inspecting the program, manually decomposing the control flow graph and/or adding conditions in order to get out of the infeasibility region.

3.2 Software Testing and Supervised Learning

In order to support Statistical Structural Software Testing, one possibility is to use supervised 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 on the CSP, filtering out the paths that are deemed infeasible and thus significantly reducing the 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 hundred 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 many orders of magnitude). Lastly, the 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. Indeed Variable Order Markov Models could accommodate such patterns [BEYY04]; however 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. This limitation is addressed through a frugal and flexible representation inspired by Parikh maps, first presented in [BSGG07].

3.3 Extended Parikh representation

Parikh maps [HU79] characterize a string from its histogram with respect to alphabet Σ; to each symbol v in Σ is associated an integer attribute a_v, counting the number of v occurrences in every string.

As this representation is clearly insufficient to account for long range dependencies in the strings, additional attributes are defined. For each pair (v, i) in $\Sigma \times \mathbb{N}$, attribute $a_{v,i}$ is defined as follows; to each string s in Σ^* it associates the successor of the i-th occurrence of the v symbol in s, or v_f if the number of v occurrences in the string is less than i.

$$\begin{aligned} v \in \Sigma &\rightarrow a_v : \Sigma^* \mapsto \mathbb{N} \\ (v, i) \in \Sigma \times \mathbb{N} &\rightarrow a_{v,i} : \Sigma^* \mapsto \Sigma \\ For\ s \in \Sigma^* a_v(s) &= |\{t_i, s[t_i] = v, t_i < t_{i+1}\}| \\ a_{v,i}(s) &= s[t_i + 1] \text{ or } v_f\ if\ i > a_v(s) \end{aligned}$$

Table 1. Extended Parikh representation

The size of this propositional representation is $|\Sigma| \times k$ where $k << T$ is the maximal number of occurrences of any symbol in a T-length string.

[1] In all generality, three classes should be considered (feasible, infeasible and undecidable) as the underlying constraint satisfaction problem is undecidable. However the undecidable class depends on the constraint solver and its support is negligible in practice.

However, while the extended Parikh representation decreases the gap between the complexity of the instances and the number of available training examples, the number of training examples is still insufficient to enable supervised learning.

In summary, the use of discriminant ML to support statistical structural software testing faces a bootstrap problem: ML requires more feasible paths; but more feasible paths is all what SSST requires, too. It thus comes to recast the discriminant ML task as an active learning task, oriented toward the generation of new feasible paths. We only need to keep in mind that active learning is severely impaired when dealing with tiny concepts [Das05].

4 Overview

This section describes the *S4T* system aimed at the generation of new feasible paths, composed of three modules. The Init module constructs a maximally specific (disjunctive) description of the initial feasible paths (the S set, in terms of Version Space). The Constrained Exploration module achieves the generation of paths subject to some constraints. The Generalization module on one hand generalizes the S set based on the new feasible paths, and on the other hand provides the Constrained Exploration module with new constraints, focussing the exploration of the search space. All three modules interact with the Oracle module (the CSP solver), labelling every new path generated as *feasible* or *infeasible.*

4.1 Init Module

With respect to the Parikh representation, the feasible path target concept is a small disjunct concept [HAP89]: the conjunction of XOR and Loop patterns is rewritten as a disjunction of conjuncts noted $C_1 \vee \ldots \vee C_K$.

The Init module is a two step process, first determining for every pair of feasible paths whether they can belong to the same conjunct, and thereafter constructing a maximally specific description of every conjunct represented in the training set. The identification of other conjuncts is left for further study.

The first step of the Init module is based on the following proposition. If two feasible paths s and s' belong to the same C_i, then their least general generalization $lgg(s, s')$ is correct, i.e. it does not cover any unfeasible path; if s and s' do not belong to the same C_i, then an example generated in $lgg(s, s')$ will be unfeasible with high probability, for the C_i's have tiny coverage.

Accordingly, a stochastic approximation of the predicate "s and s' belong to the same C_i", noted $\mathcal{R}(s, s')$, is implemented (Table 4.1.a). This approximation calls the Constrained Exploration module to independently generate and label p paths in $lgg(s, s')$. If all p paths are feasible, $\mathcal{R}(s, s')$ returns true, otherwise it returns false and the infeasible paths are added to $\mathcal{E}^-$. It is clear that $\mathcal{R}(s, s')$ implements a complete but incorrect approximation of the predicate *s and s' belong to the same conjunctive sub-concept*, and the incorrection probability exponentially decreases with p; a typical value for p in the experiments (section 5) is $p = 2$.

(a) Routine $\mathcal{R}(s, s')$

```
If ($lgg(s, s')$ covers an unfeasible path)
    return False
For $i = 1$ to $p$
    $s''$= Exploration ($lgg(s, s')$)
    If (label($s''$) = unfeasible)
        return False
Return True
```

(b) Routine Clique(s)

```
$S_0(s) = \{s\}$
$t = 1$
$\mathcal{V}_t = \{s'/\mathcal{R}(s', s'') for\ all\ s'' \in S_{t-1}(s)\}$
While $\mathcal{V}_t$ is not empty
    $s' = argmax_{\mathcal{V}_t}\{|\{s''\ in\ \mathcal{V}_t/\mathcal{R}(s', s'')\}|\}$
    $S_t(s) = S_{t-1}(s) \cup \{s''\}$
    $t \leftarrow t + 1$
Return $S_t(s)$
```

Table 2. The Init Module

In a second step, the Init module extracts maximal cliques from the graph defined from the set $\mathcal{E}^+$ of the initial feasible paths, and the $\mathcal{R}$ relation. For each path s in $\mathcal{E}^+$ (not already covered by a clique), the maximal clique $S(s)$ containing s is greedily and iteratively constructed as follows (Table 4.1.b). Let $S_0(s) = \{s\}$. At each step $t > 0$, let $\mathcal{V}_t(s)$ denote the set of elements related by $\mathcal{R}$ to all elements of $S_t(s)$. If $\mathcal{V}_t(s)$ is empty, stop; otherwise, determine the element s' in $\mathcal{V}_t(s)$ which is related by $\mathcal{R}$ to the most elements of $\mathcal{V}_t(s)$ (ties are randomly broken); add s' to the clique ($S_{t+1}(s) = S_t(s) \cup \{s'\}$).

Finally, the Init module produces a set of cliques noted $\hat{C}_i$; every feasible path in $\mathcal{E}^+$ belongs to at least one such clique. By abuse of notations, $\hat{C}_i$ is both viewed as a set of feasible paths and their lgg.

It is shown that with high probability, for every target conjunct C_i represented in $\mathcal{E}^+$ there will be some $\hat{C}_i$ such that $\hat{C}_i$ is a specialization of C_i; the probability exponentially increases with the number of representatives of C_i in $\mathcal{E}^+$ (proof omitted due to space limitation).

4.2 Generalization Module

The Generalization module aims at maximally generalizing every $\hat{C}$ produced by the Init module; it proceeds by generating new paths s "close" to $\hat{C}$ and using them to generalize $\hat{C}$ if these are labelled feasible.

Two generation procedures are considered. The first one, referred to as ϵ-Greedy generalization, is based on decorating the FSA (section 3.1) with probabilities, alternatively exploiting these probabilities and updating them after the current path has been labelled. The second one, referred to as Near-Miss-based generalization, exploits the unfeasible paths close to $\hat{C}$.

ϵ-Greedy generalization. Formally, let s denote the path under construction (initialized to the start symbol), let v denote the last symbol in s and assume that the number of v occurrences in s is i ($a_v(s) = i$). Let w denote a possible successor of the v node and let j denote the current number of occurrences of w in s ($a_w(s) = j$).

Ideally, the next node in s is selected in order to maximize the probability for s to be feasible ($a_{v,i}(s) = argmax_w\{Pr(sw * feasible)\}$ where $sw*$ stands for any path with prefix sw). However due to the sparsity of the available examples, such probabilities cannot be estimated accurately. We therefore associate to each possible successor w of the last node v the fraction p_w of feasible paths, among all paths s' in $\mathcal{E}$ having w as successor of the i-th occurrence of symbol v, and with at least $j+1$ occurrences of w ($p_w = Pr_{emp}(s'\ feasible|\ [a_{v,i}(s') = w] \wedge [a_w(s') > j])$). If p_w is defined for all successors w of the current node, the ϵ-Greedy generalization selects the next node w that maximizes p_w. Otherwise, (there exists some successor w that was never encountered as successor of the i-th occurrence of v, neither for the feasible nor for the unfeasible paths), w is selected with probability ϵ. Other heuristics enforcing a more sophisticated exploration vs exploitation trade-off, e.g. based on the multi-armed bandit UCB algorithm [ACBF02] were also considered; but they are hindered as the reward probability is very low (being reminded that the fraction of feasible paths commonly is below 10^{-5}).

Near-Miss-based generalization. Let $\hat{C}$ denote the current clique considered. Notably, $\hat{C}$ induces a partial ordering $<_C$ on the paths, defined as $s <_C s'$ iff $lgg(C \cup \{s\}) \prec lgg(C \cup \{s'\})$ where $A \prec B$ is meant for A is more specific than B in the extended Parikh representation.

Among the paths that are minimal after the above order relation, a specific case is that of unfeasible paths which differ from $\hat{C}$ by a single attribute[2]. Other minimal unfeasible paths are referred to as nearest-miss examples. For every nearest-miss example s, the Constrained Exploration module is required to generate examples in $lgg(\hat{C} \cup \{s\}) - lgg(\hat{C})$. The generated examples are labelled; if they are feasible, $\hat{C}$ is generalized; otherwise, they are used to update the set of near-miss.

4.3 Constrained Exploration module

Given a set of paths E and a set of constraints expressed in the extended Parikh representation, the constrained generation module aims to generate a path s which satisfies the constraints, noted $c(s)$.

Two cases are distinguished. In the first case, referred to as explicit, the constraints can be expressed by specializing the FSA (section 3.1) describing the path search space. In this case, the uniform sampling of the T-length paths based on the FSA can achieved analytically [FZC94].

In the second and most frequent case, referred to as implicit, the constraints are expressed using the Parikh representation and they cannot be expressed analytically within the FSA: ensuring that a given path in the FSA will satisfy these constraints boils down to solving a CSP. In the implicit case, the ϵ-Greedy generation procedure above is extended to account for the constraints $c(s)$.

[2] Such unfeasible paths, referred to as near-miss examples, signal that the single discriminant attribute must not be generalized [Mit82].

5 Experimental Validation

This section presents our experimental setting and goals, and reports on the results of *S4T*.

5.1 Experimental Setting

S4T is first validated on the real-world Fct4 problem, including 36 nodes and 46 edges (Fig. 1). The ratio of feasible paths is circa 10^{-5} for a maximum path length $T = 250$.

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 given path in the above graph is feasible. After section 3, the target concept is a conjunction of XOR concepts and Loop conditions. 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 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 conjunction.

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 $\mathcal{E}$ composed of 50 feasible and 50 infeasible paths[4]. For each conjunct $\hat{C}$ identified, the ϵ-Greedy or Near-miss generalization module is launched 400 times; the new distinct feasible paths are gathered in $\mathcal{E}^*$.

The algorithm performance is assessed by comparing for each conjunct C of the target concept represented in the training set, its initial and final coverage, that is, the fraction of paths covered by C that respectively belong to $\mathcal{E}$ and $\mathcal{E} \cup \mathcal{E}^*$, noted $i(C)$ and $f(C)$. For a better visualization, the average final coverage is computed using a Gaussian convolution: $f(x) = \frac{\sum_{C \cap \mathcal{E} \neq \emptyset} f(C) exp(-\kappa(x - i(C))^2)}{\sum_{C \cap \mathcal{E} \neq \emptyset} exp(-\kappa(x - i(C))^2)}$

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

The goal of the experiments is firstly to see whether *S4T* can efficiently sample the conjuncts that are represented in the initial training set, and how the

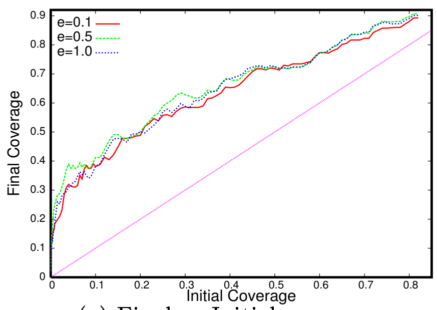
[3] Three non-terminal nodes were considered (the generic structure B, the *if* and the *while* structures), together with two terminal nodes (the *Instruction* and the *Condition* node. The probabilities on the production rules control the length and depth of the control flow graph. Eventually, the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

[4] Increasing the number of infeasible training paths does not make any difference, as only infeasible paths "close" to the feasible ones convey useful information.

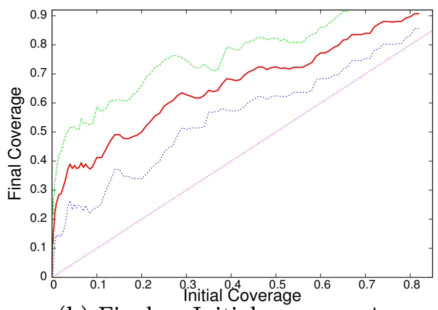
efficiency depends on the initial coverage of the conjunct in the training set. The second goal is to compare the two ϵ-greedy and Near-Miss based generalization procedures.

5.2 ϵ-greedy *S4T*

Fig. 2.(a) displays the final vs initial coverage provided by *S4T* on 10 artificial problems, using the ϵ-Greedy generalization module with $\epsilon = .1, .5$ and 1. The detailed results with standard deviation are reported on Fig. 2.(b) for $\epsilon = .5$. These results show that *S4T* efficiently samples the conjuncts that are represented in the training set. More detailed results are presented in Table 3; when the initial coverage of the conjunct 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%. For conjuncts which are already well represented in the initial training set, the gain can only be moderate.



(a) Final vs Initial coverage for $\epsilon = .1, .5$ and 1



(b) Final vs Initial coverage $\pm\sigma$ (b) for $\epsilon = .5$

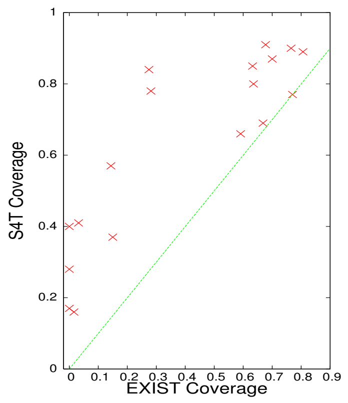
Fig. 2. *S4T* with ϵ-Greedy generalization. Final vs Initial conjunct coverage, average results on 10 artificial problems $\times$ 10 runs.

	$[0, 10^{-4}]$	$[10^{-4}, 10^{-3}]$	$[10^{-3}, 10^{-2}]$	$[10^{-2}, 10^{-1}]$	$[.1, .3]$	$[.3, .6]$	$[.6, 1]$
$log(f/i)$	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 3. Gain obtained with ϵ-greedy generalization for various ranges of the initial coverage of the conjunct.

The Fig. 3 reports the gain obtained on the real-world fct4 problem comparatively to [BSGG07] (EXIST algorithm) for 10 independent runs for an identical number of generated paths (around 3.000). The gain 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 runs of the generalization module (excluding labelling cost).



Initial coverage	EXIST Final Coverage	S4T Final Coverage
(0, .03)	0.01 ± 0.01	.25 ± 0.1
(.09, .13)	0.1 ± 0.06	.45 ± 0.07
(.21, .39)	0.44 ± 0.16	.78 ± 0.07
(.49, .52)	0.71 ± 0.05	.83 ± 0.07

Fig. 3. *S4T* with ϵ-Greedy generalization ($\epsilon = .5$) vs EXIST algorithm, average results on 10 runs on FCT4.

5.3 Near-Miss *S4T*

In contrast, the Near-Miss variant of *S4T* did not provide satisfactory results, for the following reason. As noted in section 4.2, near-miss unfeasible paths s only signal that the single attribute discriminating s from the current conjunct $\hat{C}$ should not be generalized [Mit82]. For this reason, only nearest-miss unfeasible paths were used to guide the Constrained Exploration module. However, it turns out that the Constrained Exploration module fails to construct examples in $lgg(\hat{C} \cup s) - \hat{C}$.

This failure is explained as the attributes in the extended Parikh representation are not independent: selecting one successor node instead of another one usually entails other consequences (e.g. increasing the number of occurrences of another node). For this reason, most nearest-miss examples are actually near-miss, in the sense that they are maximally close to the current conjunct: $lgg(\hat{C} \cup s) - \hat{C}$ is empty.

Therefore, the Near-Miss generalization module should rather use unfeasible paths that are sufficiently "far" from $\hat{C}$. Preliminary results along this line show convincing improvements, although it remains to adjust the appropriate Hamming distance between the useful unfeasible examples and the current $\hat{C}$.

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 [CFW06]), in software testing and in other structured domains.

The second contribution of the presented work is to construct a distribution on the top of this representation, enabling the active sampling of desired paths. Active Learning, a hot topic in the Machine Learning field for over a decade [CGJ95], is convincingly motivated by the cost of example labelling and the abundance of unlabelled examples in quite a few application domains. However, in other domains such as Numerical Engineering, examples must be constructed on purpose and their construction is expensive. The ability of biasing the example construction in order to satisfy desired properties, might thus open new application perspectives to Relational Machine Learning.

With respect to Statistical Software Testing, the presented approach dramatically increases the ratio of (distinct) feasible paths generated, compared to the former uniform sampling approach [DGG04]. Further research is concerned with sampling conjuncts 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 [Yan04] will be studied.

References

[ACBF02] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[BEYY04] R. Begleiter, R. El-Yaniv, and G. Yona. On prediction using variable order markov models. *JAIR*, 22:385–421, 2004.

[BGC01] L. Bréhélin, O. Gascuel, and G. Caraux. Hidden markov models with patterns to learn boolean vector sequences and application to the built-in self-test for integrated circuits. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(9):997–1008, 2001.

[BSGG07] N. Bastiokis, M. Sebag, M.-C. Gaudel, and S.-D. Gouraud. Software Testing: A Machine Learning Approach. In *IJCAI*, pages 2274–2279, 2007.

[CFW06] A. Clark, C. C. Florencio, and C. Watkins. Languages as hyperplanes: Grammatical inference with string kernels. In *ECML, to appear*, 2006.

[CGJ95] David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan. Active learning with statistical models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 705–712. The MIT Press, 1995.

[Das05] S. Dasgupta. Coarse sample complexity bounds for active learning. In *NIPS*, pages 235–242, 2005.

[DGG04] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *ISSRE*, pages 25–34, 2004.

[ECGN99] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, 1999.

[FZC94] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994.

[HAP89] R.C. Holte, L.E. Acker, and B.W. Porter. Concept learning and the problem of small disjuncts. In *Proceedings of IJCAI-89*, pages 813–818. Morgan Kaufmann, 1989.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Mit82] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[RDTK06] I. Rish, R. Das, G. Tesauro, and J. Kephart. *ECML-PKDD Workshop* Automatic Computing: A new Challenge for Machine Learning. 2006.

[VSVA04] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for FIFO automata. In *FSTTCS*, pages 494–505, 2004.

[XSHW05] G. Xiao, F. Southey, R. C. Holte, and D. F. Wilkinson. Software testing by active learning for commercial games. In *AAAI*, pages 898–903, 2005.

[Yan04] M. Yannakakis. Testing, optimization, and games. In *ICALP*, pages 28–45, 2004.

[ZJL$^+$06] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML*, pages 1105–1112, 2006.