# CARE: a platform for reliable Comparison and Analysis of Reverse-Engineering techniques

Sylvain Lamprier, Nicolas Baskiotis, Tewfik Ziadi and Lom Messan Hillah
*University Pierre et Marie Curie*
*LIP6 - UPMC*
*Paris, France*
*Email: surname.name@lip6.fr*

***Abstract*—Reverse engineering of behavior models has received a lot of attention over the last few years. However, no standard benchmark exists for the comparison and analysis of published miners. Evaluation is usually performed on few case studies, which fails to demonstrate effectiveness in a broad context. This paper proposes a general, approach-independent, platform for the intensive evaluation of behavior miners. Its goals are essentially: provide a benchmark mechanism for reverse engineering; allow analysis of miners w.r.t. a class of programs and/or behaviors; help users in choosing the best suited approach for their objective.**

***Keywords*-Reverse engineering; Inference mechanisms; Software maintenance; Reasoning about programs; Artificial intelligence; Artificial program generation; Evaluation; Benchmarking**

## I. Introduction

Behavior models, such as Finite State Automata (FSA), use a state representation to model the flow of the execution of a system. They play an important role in the traditional engineering of software-based systems; it is the basis for systematic approaches to design, simulation, code generation, testing, and verification. Nevertheless, such models are often not maintained during the development phase or, when it is the case, there is no guarantee about their consistency, regarding for instance completeness (all behaviors of the system are represented in the model) and correctness (behaviors described by the model belong to the system).

When behavior models are either absent or inconsistent, reverse engineering techniques can be used to extract them Interesting surveys on the reverse engineering of FSA can be found in [1], [2]. Mainly two approaches have been developed: while static analysis focuses on source code, dynamic analysis considers traces obtained from executions of the corresponding program. For object-oriented systems, dynamic analysis is best suited to extract the behavior model. It allows capturing runtime-induced characteristics of the program, like polymorphism and dynamic bindings [4]. Moreover, dynamic analysis can be instrumented even in the absence of the program source code. We thus focus in this paper on dynamic reverse engineering techniques.

Most of recent researches have mainly focused on particular contexts (test, verification, program comprehension), which induces important difficulties for evaluation purposes. Comparisons between FSA inference algorithms (also referred to as miners in the following) are nevertheless possible, as the different objectives can be expressed by the use of dedicated evaluation measures, adapted to the context.

Moreover, as mentioned in [3], previously published behavior miners lack suitable benchmarks for their evaluation. Usually, evaluation is performed on standard case studies, which is useful for assessing the feasibility of the algorithm on particular cases, but fails to demonstrate the effectiveness of the miner in a broader context.

In this paper, we propose CARE[1], a standard platform for the evaluation of behavior miners. CARE is approach-independent: the only need for the miner to be tested is to accept a set of traces as input and produce an FSA model. It includes an artificial data (programs + traces) generation tool for benchmark purposes, which is designed for modeling different classes of programs (groups of programs sharing similar properties) and behavioral aspects. The goals of CARE are essentially: provide a benchmark mechanism for reverse engineering; allow analysis of miners w.r.t. to a class of programs and/or behaviors; help users in choosing the most accurate approach w.r.t. their objective.

The remainder of the paper is organized as follows: Section II presents the generation process of data and Section III reports and interprets experimental results obtained for demonstration purposes. Finally, Section IV sketches the perspectives of this work.

## II. Generation of Artificial Data

After giving details on how programs are specified in the platform, this section presents both processes of artificial programs generation and simulated traces collection.

### A. Program Specification

Programs are specified by nested blocks determining their structure. Our program specification follows a formal grammar $G = (N, \Sigma, P, S)$, where $N$ is the set of non-terminal symbols, $\Sigma$ the set of terminal symbols, $S$ the

[1]The CARE platform is available at http://care.lip6.fr

starting symbol $Block$ and $P$ the following set of production rules:

$$\begin{aligned} Block &\rightarrow BlockList \mid Alt \mid Opt \mid Loop \mid Call \quad (1) \\ BlockList &\rightarrow Block^{+} \\ Alt &\rightarrow Block \quad Block^{+} \\ Opt &\rightarrow Block \\ Loop &\rightarrow Block \\ Call &\rightarrow MethodLabel \quad Block^{?} \quad MethodLabel \end{aligned}$$

$BlockList$ is a succession of program blocks; $Alt$ corresponds to an alternative between at least two program blocks; $Opt$ stands for an optional part of the program; $Loop$ corresponds to a part of program that can be repeated several times; $Call$ corresponds to the call of a given method, which owns method labels (opening and closing) and possibly produces a child block (allowing methods imbrication).

Such a specification presents two strong advantages: Behavior traces may be extracted from the structure (by simulating program executions, see section II-C); A corresponding FSA may be extracted to stand as a reference model for evaluation metrics.

## B. Generation of Programs

The program generation process follows two main steps: **1. Vocabulary generation:** The vocabulary is generated by determining a given number of objects and a given average number of methods names per object. An object is arbitrarily chosen to be the main one (starting point). **2. Structure construction:** The structure is recursively built by imbrication of blocks of different kinds, which are chosen according to a given distribution of probabilities. In order to get more realistic programs that may have similar behaviors in different contexts, the process may choose existing blocks, which correspond to blocks that have already been built and are consistent with the current branch of the structure (especially with respect to the current active object). Methods of Call blocks are chosen among available methods for a randomly selected callee.

Table I reports 7 different distributions of probabilities for block selection. These distributions, used in experiments presented below, imply different levels of difficulty for FSA extraction from execution traces: while some configurations lead to structures containing neither alternatives nor loops, others favor the building of complicated structures. Such configurations determine different classes of programs. Configuration A leads to linear structures, configuration C to acyclic structures with alternative paths, configuration E to structures with cycles but no alternations and configuration G to more complex structures mixing loops structures and alternations. Configurations B, D and E lead to similar automata (but with repeated patterns which complicates the problem of FSA inference from traces).

| Param \ Config | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| P(blocklist) | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| P(alt) | 0 | 0 | 0.1 | 0.1 | 0 | 0 | 0.1 |
| P(opt) | 0 | 0 | 0.1 | 0.1 | 0 | 0 | 0.1 |
| P(loop) | 0 | 0 | 0 | 0 | 0.2 | 0.1 | 0.1 |
| P(call) | 0.9 | 0.7 | 0.7 | 0.5 | 0.7 | 0.6 | 0.5 |
| P(existing) | 0 | 0.2 | 0 | 0.2 | 0 | 0.2 | 0.1 |

Table I
EXPERIMENTED PROBABILITY DISTRIBUTIONS

## C. Trace Extraction

Once the structure of the program is defined, traces are obtained by traversing it, following paths and reporting method labels, until reaching the end of the program. Extracting traces may differ thanks to the different output alternatives of branching blocks *Alt*, *Opt* and *Loop*. Different strategies of structure traversal may then be defined. According to the strategy, extracted paths may well represent the whole program behavior or only give a restricted view of all possibilities. Investigated strategies are: **i) Uniform:** every possible trace gets the same probability of belonging to the observations set, independently from its length or the number of branching blocks its path traversed[2]; **ii) Unbiased:** executions are simulated by traversing the structure with uniform choices at every branching block; **iii) Biased:** a distribution of probabilities has been set on every branching block of the structure; **iv) StateVector:** at every branching point, the choice of a given branch should depend on the current state of the program (which includes inputs and actions that have been performed since the start of the program).

## III. EXPERIMENTS/USE CASES

Several FSA extraction algorithms from positive examples have been included in the CARE platform, some of them coming from the field of grammatical inference, others being specifically designed for reverse engineering: some are mainly heuristics (KTail, Temporal KTail), others rely on statistical learning techniques (Markov Model for reverse engineering). Nevertheless, for the sake of simplicity, experiments reported here mainly focus on the classical KTail algorithm [5].

The central idea of this algorithm is to consider a state of the automaton according to the future behaviors that can been observed from it. After an FSA initialization step in which each trace from the observations set leads to the production of a specific output branch of the initial

[2]For computational purposes, we arbitrarily consider 100 as the maximal length of traces drawn from uniform strategy.

state, KTail merges all states that share the same future. Rather than considering the whole future of states (which would prevent merging when loops occur), KTail assumes that the future of a state can be characterized by the k next statements of paths traversing it. Merging two states then corresponds to assuming a generalization hypothesis: executions sharing the same k next statements will follow a same future behavior. Parameter k then allows to control the specialization/generalization level of KTail: with lower values of k, the merging process is less constrained. Hence, the resulting FSA can be more general than with higher values.

Thanks to this parameter, KTail appears well suited for our experiments, whose goal is not to compare state-of-the-art algorithms, but rather to present some use cases of the platform in order to demonstrate its usability and relevance.

### A. *Evaluation Process and Criteria*

In past evaluation competitions (e.g. in the Grammar Inference domain or in the context of software engineering), participants were given a set of strings, labeled as positive or negative (according to their occurrence in the reference model), and the task boiled down to a binary classification task. Models were then evaluated according to a sensitivity criterion, which considers the ratio of positive strings that have actually been classified as positive, and a sensibility criterion, which corresponds to the proportion of negative sequences that have been classified as so.

While our platform is objective-independent, we chose to focus in our experiments on completeness and correctness of the resulting automata. As in [6], we therefore consider recall and precision measures, which correspond to classical measures from the Information Retrieval domain. Recall renders completeness by considering the ratio of paths from the reference model that are accepted by the hypothetical one. Precision renders correctness as the ratio of statements sequences from the hypothetical automaton that belong to the reference one. Given the huge number of possible paths in the automata (possibly infinite), such measures cannot be exactly computed. Scores may nevertheless be accurately approximated by defining subsets of paths uniformly chosen among the whole set of possible paths (from the reference or the hypothetical model, according to the considered measure), and computing ratios on these subsets. In the following, we work with subsets of 1000 sequences for both measures.

### B. *Example of Algorithm Analysis*

This section aims at giving an idea of which kind of analysis may be made, with artificial programs, thanks to the proposed platform. For this example, we focus on the problem of setting an optimal size $k$ for future behaviors considered in KTail. We performed evaluation of KTail for different values of $k$ over 1000 artificial programs for each of the seven program classes defined in Table I. Observations sets used by KTail contain each 100 traces extracted following strategies defined in the previous section. Table II gives average results obtained with uniform extraction of paths for each class of program. As expected, while for larger values of $k$ resulting FSA get better precision scores (less mergers, the FSA is more specific), with low $k$ recall is favored.

Programs from class A and B are linear. The whole set of possible behaviors is then observed (only one trace is possible), which results in a perfect level of recall for every algorithm. Programs from class B nevertheless contain repetitions of sub-programs (existing blocks), which may be considered as loops by KTail (and induce an important loss of precision) when mergers are not sufficiently constrained (considering linear programs, any merge introduces generalization errors).

With programs from class C and E, which respectively contain alternatives or loops, being able to merge states is of great importance to detect such branching points. It allows to reduce the size of the produced FSA and to generalize from observations to infer paths that have not been observed but belong to the behavior model of the program. In both cases, $k = 3$ appears to allow the best compromise between completeness and correctness of the produced model, for a reasonably sized FSA.

Programs from classes D and F respectively own similar structure than those from classes C and E but contain repetitions of program blocks. We note a significant loss of performance compared with results over classes C and E, particularly for runs with low values of $k$. Same manner as for class B, repetitions of blocks in the structure introduce important generalization errors. Same tendencies are observed with programs of class G, which corresponds to the most complex set of programs (every kind of block is possible, much greater number of possible paths).

Two values for $k$ are highlighted: while $k = 5$ allows to limit generalization errors, $k = 3$ appears to obtain rather good accuracy for reasonable sized automata. Figure 1 shows the $F1$ scores[3] obtained with $k$ values 3, 5 and 1000 for all program classes and extraction strategies. Differences between scores of the three considered algorithms are not constant over the various kinds of observations sets for each program class. With program class F for instance, while $KTail\_1000$ obtains a $F1$ score similar to the one of $KTail\_3$ when observations are very restricted, it increases much greater with better views of the whole behavior. More interestingly, we may note that, when blocks cannot be repeated in the structure, the dominance of $k = 3$ is largely greater for restricted views of the whole behavior (see for example program class E, where score difference between $k = 3$ and $k = 5$ is around 0.1 with $StateVector\_0.2$,

[3]$F1$ is a classical measure allowing to mix recall and precision in a single score : $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$

| Algo \ Class | A | | | B | | | C | | | D | | | E | | | F | | | G | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | R | P | S | R | P | S | R | P | S | R | P | S | R | P | S | R | P | S | R | P | S |
| Ktail_0 | 1.00 | 0.00 | 1.00 | 1.00 | 0.00 | 1.00 | 1.00 | 0.00 | 1.00 | 1.00 | 0.00 | 1.00 | 1.00 | 0.00 | 1.00 | 1.00 | 0.00 | 1.00 | 0.99 | 0.00 | 1.00 |
| Ktail_1 | 1.00 | 0.70 | 32.59 | 1.00 | 0.29 | 25.08 | 0.99 | 0.52 | 67.93 | 0.99 | 0.16 | 50.84 | 1.00 | 0.73 | 30.02 | 1.00 | 0.33 | 21.43 | 0.98 | 0.30 | 49.86 |
| Ktail_2 | 1.00 | 0.90 | 33.35 | 1.00 | 0.30 | 27.49 | 0.99 | 0.70 | 81.53 | 0.99 | 0.17 | 68.64 | 1.00 | 0.88 | 36.53 | 1.00 | 0.34 | 25.80 | 0.97 | 0.34 | 73.32 |
| Ktail_3 | 1.00 | 1.00 | 33.49 | 1.00 | 0.55 | 29.32 | 0.98 | 0.99 | 93.64 | 0.98 | 0.38 | 87.53 | 1.00 | 1.00 | 43.23 | 1.00 | 0.60 | 30.02 | 0.94 | 0.61 | 100.0 |
| Ktail_4 | 1.00 | 1.00 | 33.49 | 1.00 | 0.55 | 30.24 | 0.98 | 1.00 | 106.1 | 0.97 | 0.39 | 106.5 | 0.99 | 1.00 | 52.25 | 1.00 | 0.60 | 34.31 | 0.91 | 0.62 | 142.1 |
| Ktail_5 | 1.00 | 1.00 | 33.49 | 1.00 | 0.75 | 31.07 | 0.97 | 1.00 | 119.7 | 0.96 | 0.61 | 127.3 | 0.99 | 1.00 | 62.81 | 1.00 | 0.76 | 39.11 | 0.85 | 0.76 | 200.1 |
| Ktail_10 | 1.00 | 1.00 | 33.49 | 1.00 | 0.92 | 32.48 | 0.93 | 1.00 | 204.2 | 0.90 | 0.82 | 251.7 | 0.91 | 1.00 | 178.5 | 0.98 | 0.87 | 80.56 | 0.63 | 0.88 | 729.9 |
| Ktail_1000 | 1.00 | 1.00 | 33.49 | 1.00 | 1.00 | 33.32 | 0.77 | 1.00 | 1404 | 0.72 | 1.00 | 1680 | 0.48 | 1.00 | 3248 | 0.73 | 1.00 | 2020 | 0.32 | 1.00 | 3877 |

Table II
Results for KTail with Uniform observations

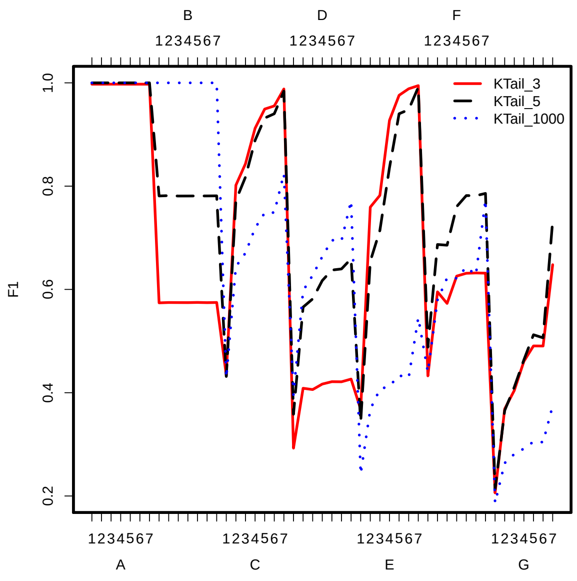

Figure 1. F1 scores for KTail with $k$ set to 3, 5 and 1000. $KTail\_3$, $KTail\_5$ and $KTail\_1000$. Letters correspond to program classes and numbers to path extraction strategies ordered by increasing coverage of the whole behavior: $1 = \mathit{StateVector\_0}$, $2 = \mathit{StateVector\_0.2}$, $3 = \mathit{Biased}$, $4 = \mathit{StateVector\_0.5}$, $5 = \mathit{StateVector\_1}$, $6 = \mathit{Unbiased}$, $7 = \mathit{Uniform}$.

whereas it is only of 0.003 for *Uniform* observations). Such examples demonstrate the importance of generating different sets of observations, following various probability distributions, in order to fairly compare behavior miners.

## IV. Conclusion

By determining structures of programs as nested blocks defining their general behavior, we proposed here a way to generate large amounts of realistic artificial data, that greatly serve analysis and comparisons purposes. To the best of our knowledge, the CARE platform is the first one to propose: 1) the generation of diverse identified classes of programs; 2) various strategies of execution simulation that lead to different coverage levels of the program behavior.

Future work will include the integration of arguments in methods' invocations during the structure generation in order to be able to consider algorithms such as GKTail [7], which focuses on determining parameter intervals on transitions of the FSA. Stating the goal of behavior mining as producing coherent and readable Sequence Diagrams (SD), we are also currently working on new evaluation measures that could consider SD driven criteria on the resulting automaton (divisibility in sub process, coherence of contiguous calls, one closing per call, consistent closings order, etc...). If such measures turn out to be relevant for evaluation purposes, they also may constitute new criteria to consider in designing mining algorithms.

## References

[1] 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, Jul. 1998.

[2] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont, "Stamina: a competition to encourage the development and assessment of software model inference techniques," *Empirical Software Engineering*, pp. 1–34, 2012.

[3] N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, and P. Dupont, "A framework for the competitive evaluation of model inference techniques," in *MIIT '10*. New York, NY, USA: ACM, 2010, pp. 1–9.

[4] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Tran. on Sof. Eng.*, vol. 32, no. 9, pp. 642–663, 2006.

[5] A. W. Biermann and J. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions Computers*, vol. 21, pp. 592–597, 1972.

[6] D. Lo and S. C. Khoo, "Quark: Empirical assessment of automaton-based specification miners," in *WCRE '06*. Washington, DC, USA: IEEE CS, 2006, pp. 51–60.

[7] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *ICSE '08*. New York, NY, USA: ACM, 2008, pp. 501–510.