Information and Software Technology 60 (2015) 32–50

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

The CARE platform for the analysis of behavior model inference techniques

Sylvain Lamprier[a], Nicolas Baskiotis[a], Tewfik Ziadi[a], Lom Messan Hillah[b,*]

[a] CNRS UMR 7606 (LIP6), Sorbonne Universités, Université Pierre et Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, France
[b] CNRS UMR 7606 (LIP6), Université Paris Ouest Nanterre La Défense, 200 avenue de la République, 92001 Nanterre Cedex, France

ARTICLE INFO

Article history:
Received 19 May 2014
Received in revised form 27 November 2014
Accepted 10 December 2014
Available online 20 December 2014

Keywords:
Behavior
FSM
Inference
Benchmark
Evaluation

ABSTRACT

Context: Finite State Machine (FSM) inference from execution traces has received a lot of attention over the past few years. Various approaches have been explored, each holding different properties for the resulting models, but the lack of standard benchmarks limits the ability of comparing the proposed techniques. Evaluation is usually performed on a few case studies, which is useful for assessing the feasibility of the algorithm on particular cases, but fails to demonstrate effectiveness in a broad context. Consequently, understanding the strengths and weaknesses of inference techniques remains a challenging task.
Objective: This paper proposes CARE, a general, approach-independent, platform for the intensive evaluation of FSM inference techniques.
Method: Grounded in a program specification scheme that provides a good control on the expected program structures, it allows the production of large benchmarks with well identified properties.
Results: The CARE platform demonstrates the following features: (1) providing a benchmarking mechanism for FSM inference techniques, (2) allowing analysis of existing techniques w.r.t. a class of programs and/or behaviors, and (3) helping users in choosing the best suited approach for their objective. Moreover, our extensive experiments on different FSM inference techniques highlight that they do not behave in the same manner on every class of program. Characterizing different classes of programs thus helps understanding the strengths and weaknesses of the studied techniques.
Conclusion: Experiments reported in this paper show examples of use cases that demonstrate the ability of the platform to generate large and diverse sets of programs, which allows to carry out meaningful inference techniques analysis. The analysis strategies the CARE platform offers open new opportunities for program behavior learning, particularly in conjunction with model checking techniques. The CARE platform is available at http://care.lip6.fr.

1. Introduction

Behavior models, such as Finite State Machines (FSM), 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; they are the basis for systematic approaches to design, simulation, code generation [1,2], testing [3,4], and verification [5]. Unfortunately, such models are often not maintained during the development phase or, when it is the case, there is no guarantee about their consistency.

When behavior models are either absent or inconsistent, *inference techniques* can be used to extract them [6,7]. FSM inference has received a lot of attentions over the past years [8,7,9–13]. Surveys on existing work can be found in [7,14]. Inferring FSM can be developed using static analysis of the source code or dynamic analysis by considering traces obtained from the executions of the corresponding system. 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 [15]. Moreover, dynamic analysis can be instrumented even in the absence of the program source code.

We focus in this paper on FSM inference techniques that use dynamic analysis. The main challenge in these techniques is to infer a FSM for the system from a restricted view of its global behavior. Indeed, since it appears unrealistic to collect all possible executions traces of any studied system, inferring an exact model is intractable.

In practice, the proposed FSM techniques thus try to obtain a good approximated model that well specifies the full behavior of

* Corresponding author.

http://dx.doi.org/10.1016/j.infsof.2014.12.004
0950-5849/© 2014 Elsevier B.V. All rights reserved.

the system, regarding for instance completeness (i.e. all behaviors of the system are represented in the model), and correctness (i.e. behaviors described by the model belong to the system). This approximation makes it necessary for each FSM inference technique to propose its own evaluation framework to study the quality of the inferred model. Unfortunately, the evaluation is often performed in an ad hoc way when the proposed inference technique is only evaluated on standard case studies and with dedicated evaluation measures.

For instance, Lo et al. [16] use the traces extracted from the CVS system and the X11 Windowing Library to evaluate their FSM inference technique. Lorenzoli et al. [12] use open source software applications such Squirel, Jbref and Jedit for the evaluation step. These evaluations are useful for assessing the feasibility of the algorithm on particular cases, but fail to demonstrate the effectiveness of the miner in a broader context. In addition, comparing existing inference techniques thus requires to reproduce the same case studies, which is often difficult [17].

This paper proposes a standard platform, named CARE, for the intensive evaluation of FSM inference techniques, grounded in an artificial generation process of evaluation data. The CARE platform includes:

- An artificial program generator that randomly generates program specifications. From each program, a reference FSM is extracted. As we will see in Section 3, CARE allows generating different classes of programs. Each class gathers a set of programs sharing similar properties concerning their structure.
- Various strategies for execution simulations from programs' structures.
- An extensible evaluation framework including metrics to assess the quality of the inferred FSM with respect to the reference one.

This is a revised and extended version of a previous conference paper [18]. Compared to that earlier paper, we provide an in-depth full description of our evaluation platform, a more comprehensive survey of related work, and extensive experiments results.

The remainder of this paper is organized as follows: Section 2 introduces the core concepts of this work, and provides a background on FSM inference techniques. Section 3 presents the architecture of the CARE platform, and the different steps of its workflow, namely artificial programs generation, extraction of traces, FSM inference, and evaluation of the inference using metrics. Section 4 reports and interprets experimental results obtained for demonstration purposes. Section 5 discusses related work, and finally Section 6 sketches the perspectives of this work.

2. Background

We consider behavior model inference of object-oriented systems that extracts a Finite State Machine (FSM) from input execution traces, that we call *the set of observations*. Each trace corresponds to a sequence of actions observed during an execution of the considered system. Such actions usually correspond to method invocations, represented in traces as labels standing for openings and closings of method invocations. Method invocations are decomposed in openings and closings to allow the explicit representation of nested invocations. This is useful for instance to detect successive nested method calls.

In the following, we define the core concepts our approach builds upon to implement the CARE platform introduced in Section 3. First we define method labels, then traces as sequences of method labels, and Finite State Machines. We finally conclude the background by presenting the technical objectives of FSM inference and discuss the main motivations of the CARE platform.

Definition 1 (*Method Label*). A method label is a 5-tuple $\langle id, type, caller, method, callee\rangle$ where:

- *id* is a unique identifier of the method invocation;
- *type* determines whether the label corresponds to the opening or closing of a method invocation;
- *method* corresponds to the name of the invoked method[1];
- *caller* and *callee* respectively stand for the object that has invoked the method and the one on which the method is invoked.

Method labels in traces and FSM displayed in this paper are given in the form `caller:callee.method`.

Definition 2 (*Trace*). A Trace is a sequence of method labels $L = (l_1, l_2, \ldots, l_n)$, where $(l_i.id, l_i.type)$ is unique and: $\forall l_i \in L, l_i.type = \text{“closing”} \Rightarrow \exists l_j \in L, j\langle i \wedge l_j.id = l_i.id \wedge l_j.type = \text{“opening”}$.

Note that this paper only considers sequential traces. For concurrent systems, traces from different threads are sequentially collected in a unique execution trace. Non object-oriented systems can also be considered, as the process of such systems may be represented as a sequence of method invocations of a single object on itself.

Fig. 1 illustrates an example of two execution traces for an online shopping system. These traces show method invocations between three object instances `ihm0`, `buySys0` and `cart0` of their respective classes: `UserIHM`, `BuySystem`, and `Cart`. The first trace illustrates the execution of the online system where the user added three articles to the cart, then paid and validated the cart. The second trace shows a sequence of method invocations where the user added one article in the cart and then canceled the operation. Method labels representing method invocation closings are displayed in traces with the word *closing* preceding the name of the method.

Definition 3 (*FSM*). An FSM is a 4-tuple $\langle S, T, s_0, s_F\rangle$, where S is the set of states, T is the finite set of labeled transitions between states in S, s_0 is the initial state, and s_F is the set of final states. A transition $t \in T$ is a 3-tuple $\langle s, l, s'\rangle$, where l is the method label of the transition and $s, s' \in S^2$ are respectively its source and destination states.

Fig. 2 depicts the FSM model that specifies the behavior of the online shopping system from which traces in Fig. 1 have been obtained. Transitions in this FSM are labeled by method labels from the traces in Fig. 1. The loop on states S6, S7, S9, S8 represents the action of adding articles to the cart. Then an alternative is proposed between payment (S6, S14, S16, S15, S13) and cancelation (S6, S19, S18) of the sale.

2.1. FSM inference objectives

Numerous automated inference techniques, static or dynamic, have been proposed in the literature for behavior model extraction from execution traces [7,8,19–22]. The main challenge for dynamic inference techniques is that only a restricted view of the global behavior of the system is available [14]. Indeed, since it appears unrealistic to collect all possible executions traces of any studied system, inferring an exact model is intractable. In practice, the proposed approaches thus try to obtain good approximated models

[1] In this paper we do not consider method parameters.

```
1: ihm0:buySys0.addArticle()
2: buySys0:cart0.addToCart()
3: buySys0:cart0.closing_addToCart()
4: ihm0:buySys0.closing_addArticle()
5: ihm0:buySys0.addArticle()
6: buySys0:cart0.addToCart()
7: buySys0:cart0.closing addToCart()
8: ihm0:buySys0.closing addArticle()
9: ihm0:buySys0.addArticle()
10: buySys0:cart0.addToCart()
11: buySys0:cart0.closing_addToCart()
12: ihm0:buySys0.closing_addArticle()
13: ihm0:buySys0.payCommand()
14: buySys0:cart0.validate()
15: buySys0:cart0.closing_validate()
16: ihm0:buySys0.closing_payCommand()
```

```
1: ihm0:buySys0.addArticle()
2: buySys0.cart0.addToCart()
3: buySys0.cart0.closing_addToCart()
4: ihm0:buySys0.closing_addArticle()
5: ihm0:buySys0.cancel()
6: ihm0:buySys0.closing_cancel()
```

Fig. 1. Sample execution traces of the online shopping system example.

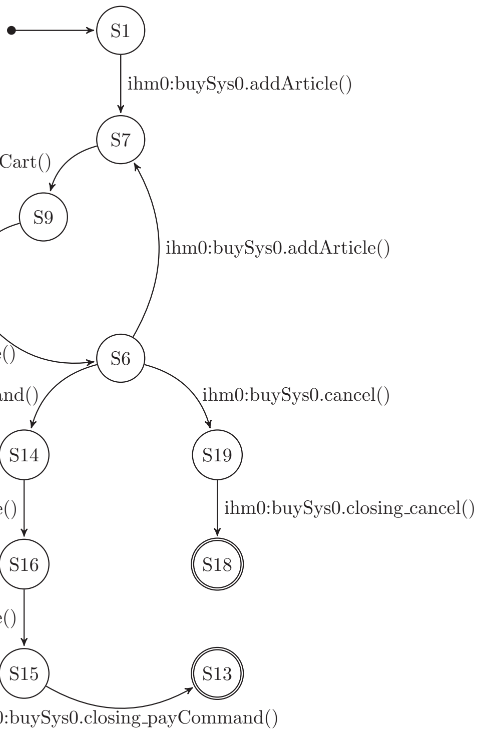
S1
ihm0:buySys0.addArticle()
S7
buySys0:cart0.addToCart()
S9
buySys0:cart0.closing_addToCart()
ihm0:buySys0.addArticle()
S8
ihm0:buySys0.closing_addArticle()
S6
ihm0:buySys0.payCommand()
ihm0:buySys0.cancel()
S14
S19
buySys0:cart0.validate()
ihm0:buySys0.closing_cancel()
S16
S18
buySys0:cart0.closing_validate()
S15
S13
ihm0:buySys0.closing_payCommand()

Fig. 2. FSM specifying the behavior of the online shopping system.

that reach a good compromise between the most specific model which only accepts the exact sequences of the observed traces, and the most general one which accepts any sequence composed by words contained in the observed traces. Fig. 3 illustrates this scale, where the most specific model is on the left-hand side, the most general on the right-hand side, and an example of expected model in the middle.

Behavior model inference targets different goals, and criteria to optimize depend on the context in which an algorithm is designed. For instance, in the context of software testing, verification or performance, the completeness and the correctness of the inferred FSM are crucial [23–25]; for program comprehension or any human usage of the inferred FSM, conciseness and readability are relevant [26].

Whatever the objective to be optimized in the resulting FSM [27], every algorithm of program modeling aims at optimizing a common risk minimization problem:

$$a* = \underset{a \in \mathcal{A}}{\operatorname{argmin}}\ R(a) \qquad (1)$$

where $\mathcal{A}$ stands for the infinite set of possible modeling algorithms, $a*$ the optimal algorithm to find and $R: \mathcal{A} \rightarrow \mathcal{R}$ a risk function defined by:

$$R(a) = \sum_{p \in \mathcal{P}} \Delta(Ref_p, Hyp_{a,p})P(p) \qquad (2)$$

with $\mathcal{P}$ the class of programs concerned by the modeling, $P(p)$ the probability to observe the program p and $\Delta: Model^2 \rightarrow \mathcal{R}$ a

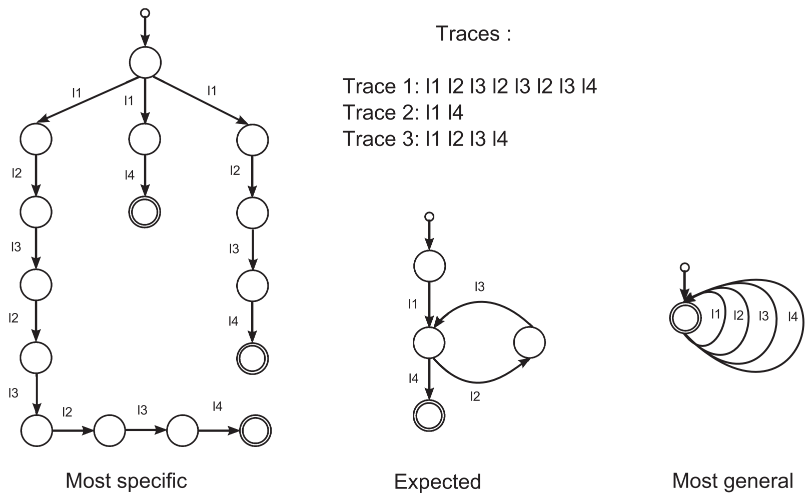

Fig. 3. From the most specific to the most general FSM.

function that measures a distance between a reference model Ref_p for the program p (what we should obtain) and the model $Hyp_{a,p}$ built by the algorithm a for this program (what we obtained). We consider here a uniform distribution of programs in $\mathcal{P}$.

Due to the size of $\mathcal{P}$ (possibly infinite), such theoretical risk cannot be computed. Classically in machine learning, such risk is then approximated by an empirical risk such as:

$$R(a) = \frac{1}{|\widehat{\mathcal{P}}|}\sum_{p\in\widehat{\mathcal{P}}} \Delta(Ref_p, Hyp_{a,p}) \tag{3}$$

where the set of possible programs of a given class $\mathcal{P}$ is replaced by a closed subset $\widehat{\mathcal{P}}$ of available programs. The gap between the theoretical risk and such empirical approximation is referred to as the learning gap; the aim of any inference technique is to build, from a given set of observations, models that are general enough to be valid on unobserved data. To be effective, the set of training observations must be as representative of the *whole world* as possible. In the context of program modeling, collecting such a set of data is a very difficult task: it boils down to a large collection of programs standing as representative examples of the class in concern. Furthermore, to perform supervised learning and evaluation of algorithms, it is not enough to get such training set of programs; it is also required to get the corresponding reference models, which is not a trivial task for most of program modeling problems.

To extract FSM from programs traces, an additional difficulty appears: we only get a restricted view of the programs' behaviors. Collecting representative sets of traces is not a trivial task either since paths followed by the system depend on several factors: inputs, interactions with users, running context, stochastic computations, etc.

The platform this paper proposes aims at enabling the work on large collections of programs, for which the reference model to reach is known and can be used for the computation of Δ functions, and for which large and diverse sets of traces are available. Beyond the fact that produced data may be very useful to design new algorithms, especially those based on machine learning techniques, this platform provides accurate tools for the evaluation and comparison of various algorithms. While in most studies evaluation is performed on a very small set of programs, typically by using handmade models designed as reference (e.g. studies reported in [7,28,29] or [30]), our platform allows to get more reliable results.

3. CARE

Fig. 4 shows an overview of the architecture of the CARE platform, which comprises four main components. Components 1, 2, and 4 are at the core of the original work described in this paper. The flow of control runs through components 1–4, in this order, with their respective inputs and outputs. The third component corresponds to the behavior inference algorithm to evaluate. It is required to accept input traces and to return an inferred FSM as output.

The distinctive features of CARE are its ability for artificial programs generation, its various traces extraction strategies from a reference FSM, and the extensible metrics and evaluation framework it offers to assess the quality of the inferred FSM with respect to the reference one. The program generator needs as input a configuration that consists of a set of probabilities over control structures appearance such as alternatives and loops, and nesting. Each generated program gets an FSM representation that is used by the traces extractor component to produce individual linear traces, each trace representing a path in the FSM, i.e. a sequence of method invocations. Inference algorithms from the literature are then applied on these traces to build a newly inferred FSM that represents the output of component 3. Finally, the metrics and evaluation framework compares the original FSM and the inferred one, and outputs different evaluation metrics rendering the proximity between these two models (see Section 3.3).

3.1. Artificial program generation

3.1.1. Program specification

As discussed earlier, there is a need in our context to deal with programs whose reference model is known. Hence, in our platform, programs are specified in a way that allows to easily get such a model. More precisely, programs are specified by nested blocks determining their structure. This structure is similar to the classical program control flow [31,32]. As we are only interested in the skeleton of a program, its general behavior and possible sequences of labels, implementation details are irrelevant to us in this setting. Our program specification follows a formal grammar $G = (N, \Sigma, P, S)$, where N is the set of non-terminal symbols, Σ the set of terminal symbols, S the starting symbol *Block* and P the following set of production rules:

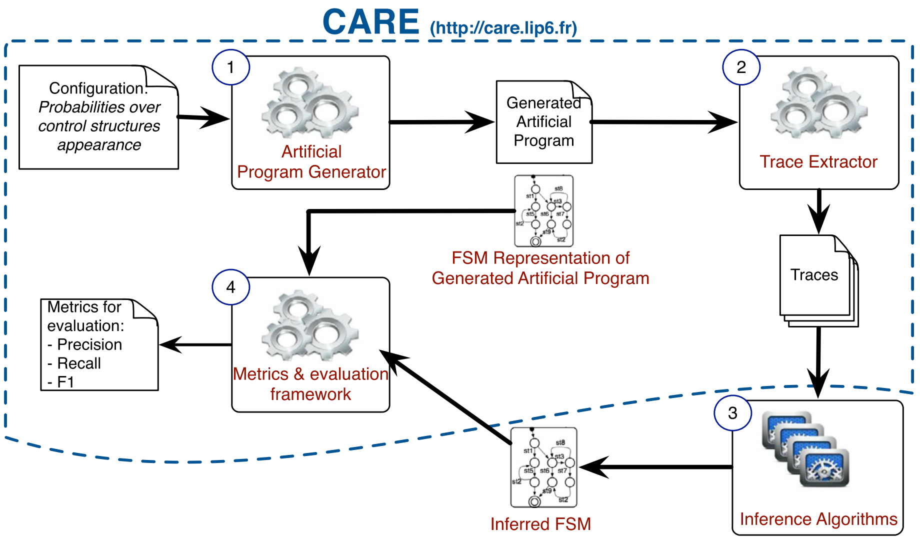
CARE (http://care.lip6.fr)
Configuration: Probabilities over control structures appearance
1
Artificial Program Generator
Generated Artificial Program
2
Trace Extractor
FSM Representation of Generated Artificial Program
Traces
4
Metrics for evaluation: - Precision - Recall - F1
Metrics & evaluation framework
3
Inference Algorithms
Inferred FSM

Fig. 4. Overview of CARE architecture.

$$
\begin{aligned}
&Block \rightarrow BlockList|Alt|Opt|Loop|Call \\
&BlockList \rightarrow Block^{+} \\
&Alt \rightarrow Block\ Block^{+} \\
&Opt \rightarrow Block \\
&Loop \rightarrow Block \\
&Call \rightarrow MethodLabel\ Block^{?}\ MethodLabel
\end{aligned} \tag{4}
$$

In this specification, each block then corresponds to one of five different types: *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 invocation of a given method, which owns method labels (opening and closing, as introduced in Section 2) and and possibly produces a child block, allowing nested blocks.

Definition 4 (*Block*). In the following, a block is defined as a 5-tuple $\langle id, type, parent, children, labels\rangle >$ where:

- *id* is a unique identifier of the block;
- *type* determines the type of the block in $\{BlockList, Alt, Opt, Loop, Call\}$;
- *parent* corresponds to the parent block of the block ($\emptyset$ if the block is at the root of the structure);
- *children* corresponds to an array of nested blocks;
- *labels* corresponds to an array of method labels (only defined for *Call* blocks);

To ensure consistent construction of blocks, the following constraint must be satisfied by every block b in the set of blocks $\mathcal{B}$ of the program in concern:

Definition 5 (*Block Construction Constraint*).

$$
\begin{aligned}
\forall b \in \mathcal{B}, (b.type = Call) \Rightarrow ((b.labels[0], b.labels[1]) \in \{(l, l') \in \mathcal{L}^2 |\ & \\
& l.caller = activeObject(b.parent) \wedge \\
& l.type = \text{“opening”} \wedge \\
& l.id = l'.id \wedge l'.type = \text{“closing”}\})
\end{aligned}
$$

where $\mathcal{L}$ is the set of available labels (which equals to the set of terminal symbols of the grammar Σ) and $activeObject(b.parent)$ refers to the current active object before entering in block b, which corresponds to the object on which the last method invocation has been performed. The active object for a given block b is equal to the callee of the closest *Call* in the parent structure (including b), or the main object if there exists no such block of type *Call* in the parent structure:

$$
activeObject(b) = \begin{cases} \text{if } b = \emptyset: & mainObject \\ \text{if } b.type = Call: & b.labels[0].callee \\ \text{Otherwise}: & activeObject(b.parent) \end{cases} \tag{5}
$$

with *mainObject* an object randomly chosen to stand as the main of the program. Beyond that it ensures the consistency of both method

```
BlocList id=1
        Alt id=2
            Call id=3 call=a:c.m1()
                Call id=4 call=c:b.m2()
                Closing Call id=4
            Closing Call id=3
            Call id=5 call=a:b.m3()
            Closing Call id=5
        End Alt
        Loop id=6
            Call id=7 call=a:b.m3()
                Opt id=8
                    Call id=9 call=b:c.m4()
                    Closing Call id=9
                End Opt
            Closing Call id=7
        End Loop
        Call id=10 call=a:b.m1()
            Loop id=11
                Opt id=12
                    Call id=13 call=b:c.m4()
                    Closing Call id=13
                End Opt
            End Loop
        Closing Call id=10
End BlocList
```

Fig. 5. Example of Block generated using CARE.

labels one with the other in each Call block, the constraint from Definition 5 implies that the caller of the method in concern well corresponds to the current active object. It guarantees consistency in message transmissions between objects, in particular well-nested *Call* blocks.

Fig. 5 shows an example of Block generated using CARE while Fig. 6 depicts its structure as a simplified abstract syntax tree. The root of the structure is a BlockList. It corresponds to a program composed of three parts that are executed sequentially, beginning by the first child branch of the root (the *Alt* block with $id = 2$) and ending by its last child branch (the Call block with $id = 10$). The Alt block ($id = 2$) defines an alternative for the program between two branches: both starting by a Call block but with different method invocations. The first alternative (the Call block with $id = 3$) illustrates the case of a Call block with a nested child block (the Call block with $id = 4$). An example of a Loop block is the one with $id = 6$: its nested branch starting from the Call block with $id = 7$ can be repeated mulitple times during the program execution. Examples of Opt blocks are the ones with $id = 8$ and $id = 12$: their nested structure is optional during execution. At last, if we consider a as the main object of the represented program (*mainObject*), we can note that the presented structure is valid with respect to the constraint Definition 5, as every caller of the method label from any Call block b corresponds to the object that was active before the execution of block b (i.e., the object returned by $activeObject(b.parent)$ such as defined in 5). First, any Call block with no ancestor of type Call in the structure indeed owns a method label with object a as its caller. Second, every other Call block indeed owns a method label with the callee of the closest ancestor in the structure as the caller: see for examples the block with $id = 4$ whose caller is equal to the callee (object c) of the method label of its parent block with $id = 3$, or the block with $id = 13$ whose caller is equal to the callee (object b) of the method label of its ancestor block with $id = 10$.

A structure such as the one presented here presents two strong advantages. First, behavior traces may be extracted from the structure, by simulating program executions, described in Section 3.2. Second, a corresponding FSM may be extracted to stand as a reference model for the evaluation metrics mentioned above. Fig. 7 shows the corresponding FSM for each kind of block of the production rules. The resulting FSM is composed by combining the FSM of nested blocks. Finally, a ϵ-removal process is applied to remove ϵ-transitions from the resulting reference model. This is done following the generic backward ϵ-removal method, which does not induce any change in the language encoded by the model [33].

3.1.2. Program generation

For each new program to build, the first step of the program generation process of CARE is a vocabulary determination step. In this step, various method labels are randomly produced by creating a given number of virtual objects and methods for these objects. For this preliminary vocabulary generation step, the two following input parameters are therefore considered:

- the number of different objects, noted *nbObj*,
- the number of method labels per object, noted *nbMethPerObj*.

The expected size of the vocabulary $|\mathcal{L}|$ is then equal to $nbObj * nbMethPerObj$. An object is finally arbitrarily chosen to be the main one (starting point), hereafter denoted as *mainObject*.

Then, once the set of possible labels is built, the program generation process follows by recursively building the program structure. As described in Algorithms 1 and 2, this is done by nesting blocks of different types according to a given probability distribution $P(t)$, with t a given type of block: for each new block to nest as a child of a given block, a type of block is sampled from this distribution $P(t)$ and a corresponding type of block is created.

However, in order to get more realistic programs that may have similar behaviors in different contexts, additionally to block types defined above (i.e., BlockList, Alt, Opt, Loop, Call) the process may choose to re-use an already existing block with a given probability $P(Existing)$. In that case, the process does not create a new block but selects an existing one in the set $\mathcal{B}$ of already entirely built blocks for the program in question (which

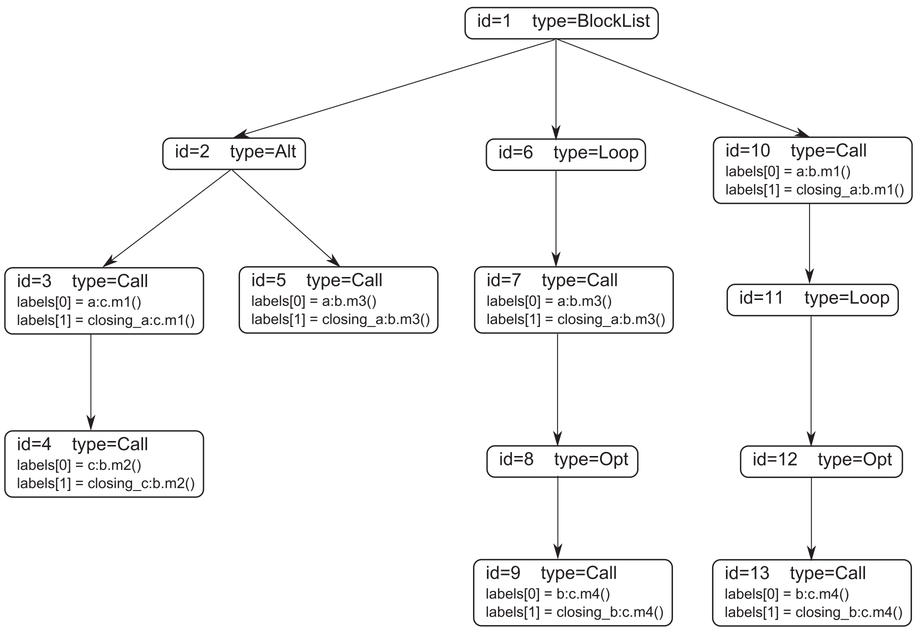

Fig. 6. Example of block structure.

contains only blocks whose nested structure is entirely defined). Note that, to respect constraint Definition 5 defined above, the selected block needs to be consistent with the active object $activeObject(b)$ of the current block b, and therefore must belong to $\{b' \in \mathcal{B} | \forall o \in nextCallers(b'), o = activeObject(b)\}$. $nextCallers(b')$ is a function returning the set of caller objects of methods invocated by the first Call child of every branch of the structure nested in b':

$$nextCallers(b) = \begin{cases} \text{if } b.type \neq Call: & \bigcup_{child \in b.children} nextCallers(child) \\ \text{if } b.type = Call: & \{b.labels[0].caller\} \end{cases} \tag{6}$$

Note that, if the structure has been correctly defined (with respect to the constraint Definition 5), the set returned by this function *nextCallers* contains a unique object. To illustrate this constraint on selected existing blocks, see Fig. 6 presented above which depicts an example of program structure. From that structure, an example of possible existing block that could be nested as a child of the Call block with $id = 5$ is the Loop block with $id = 11$, since the unique object contained in the set returned by the *nextCallers* function called on it (the object b) corresponds to the object returned by the function *activeObject* applied on the block with $id = 5$.

The generation process considers the following main configuration parameters:

- the distribution probability defined for block selection, noted $P(t)$ where $t \in \{BlockList, Alt, Opt, Loop, Call, Existing\}$;
- the maximal numbers of nested blocks in BlockList and Alt blocks, respectively noted *maxSizeBlockList* and *maxSizeAlt*,
- the probability for any Call block to have a child block, noted *Pchild*, which controls the nesting degree of the structure.

The structure construction phase starts by invoking Algorithm 1 that describes how program structures are recursively built. Note that, as described in line 13, since the algorithm is launched with a null input *parent* parameter, the root block of any program structure is a BlockList. If the process is at a deeper level of the structure, and if it chooses to re-use an existing block when sampling the type of block to define (line 17), the selected block to be re-used is deeply copied, i.e. a recursive copy is performed on its children to get a copy of the full nested structure. The copied structure is then returned; otherwise, the function CreateBlock (Algorithm 2) is called, which defines a new block of the desired type by setting required fields and calling the BuildStructure (Algorithm 1) function for each child block to determine. These recursive calls repeat until the whole structure is produced. Note that, at the end of the process, every leaf of the structure is a Call block with no child block.

3.2. Extraction of traces

Once the structure of the program is defined, traces are obtained by traversing the FSM corresponding to this program's structure, following paths and reporting method labels, from the initial state to the final state(s).

This step takes two input parameters:

- A maximal trace length parameter, noted *maxTraceLength*; traces are drawn until the required number of traces no longer than that size has been reached. Any program structure that does not allow at least one trace of length less than that maximum length is ignored;
- A strategy for structure traversal, noted $strategy \in \{Uniform, Unbiased, Biased, StateVector\}$, explained below.

Extracting traces may differ thanks to the different output alternatives of branching blocks *Alt*, *Opt* and *Loop*. Different strategies of structure traversal can be applied. According to the strategy, extracted traces may well represent the whole program behavior or only give a restricted view of all possibilities. As noted above, we propose four strategies:

Uniform: Every possible trace gets the same probability of belonging to the set of observations, independently from its length or the number of branching blocks its path traverses. The uniform strategy considering an equal probability to each sequence to be drawn, every trace is thus likely to be extracted a similar number of times, which ensures a good average representativeness of the extracted set of sequences.

Algorithm 1. Function BuildStructure

```
Global Data:(
  $BlockTypes = \{BlockList, Alt, Opt, Loop, Call, Existing\}$: possible types of blocks;
  $P(t)$: probability distribution defined over block types $t \in BlockTypes$;
  $maxSizeBlockList$: maximal number of nested blocks in BlockList;
  $maxSizeAlt$: maximal number of nested blocks in Alt;
  $Pchild$: probability for any Call block to have a child block;
  $\mathcal{L}$: the set of available method labels;
  $\mathcal{B}$: the set of already entirely built blocks for the program in concern;
  $mainObject$: the main object of the program;
)
Input:
  $parent$: parent block of the current block being constructed;
Output:
  $block$: a block, newly built or returned from existings;
if $parent = \emptyset$ then
    $\mathcal{B} \leftarrow \emptyset$;
    $block \leftarrow CreateBlock(BlockList, parent)$;
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{block\}$;
    return $block$;
end
Sample a type of block $typeBlock \sim P(t)$;
if $typeBlock = Existing$ then
    $B \leftarrow \{b \in \mathcal{B} | \forall c \in nextCallers(b), c = activeObject(parent)\}$;
    if $B = \emptyset$ then
        $block \leftarrow CreateBlock(Call, parent)$;
    else
        Sample $b$ from $B$;
        $block \leftarrow$ Deep copy of $b$;
        $block.parent \leftarrow parent$;
        return $block$;
    end
else
    $block \leftarrow CreateBlock(typeBlock, parent)$;
end
$\mathcal{B} \leftarrow \mathcal{B} \cup \{block\}$;
return $block$;
```

Algorithm 2. Function CreateBlock.

```
// Global data defined in Algorithm 1 are accessible here.
Input:
  typeBlock: a type of block;
  parent: current active object;
Output:
  block: a newly built block;
block ← create empty block, whose fields hold no value initially;
block.type ← typeBlock;
block.parent ← parent;
if $typeBlock = Call$ then
    Sample $(l, l')$ from $\mathcal{L}^2 | l.caller = activeObject(parent) \wedge$
        $l.type = \text{“opening”} \wedge l.id = l'.id \wedge l'.type = \text{“closing”}$};
    block.labels[0] ← l;
    block.labels[1] ← l';
    Sample $x$ in $[0;1[$;
    if $x < Pchild$ then
        block.children[0] ← BuildStructure(block);
    end
else
    if $typeBlock = BlockList$ OR $typeBlock = Alt$ then
        Sample $x$ in $\{2..((typeBlock = Alt)?maxSizeAlt : maxSizeBlockList)\}$;
        for $i \leftarrow 0$ to $x$ do
            block.children[i] ← BuildStructure(block);
        end
    else
        // Opt and Loop case
        block.children[0] ← BuildStructure(block);
    end
end
return block;
```

The length of each trace to extract is uniformly sampled beforehand from the set of possible lengths for the program in concern. Only lengths lower or equal to the maximal length parameter value are considered.

Unbiased: Executions are simulated by traversing the structure with uniform choices at every branching block.

Biased: A distribution of probabilities is set on every branching block of the structure. Extracted paths depend on these probabilities of branch selection.

StateVector: Considering programs as probabilistic automata is not realistic. 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. In order to better simulate real-world executions, we define:

1. An *initial state vector* noted *initialStateVector*, randomly chosen for each trace extraction, and which stands for the context of program execution (inputs, external factors, etc.).
2. An *effect vector* noted *b.effect*, randomly set for every *Call* block *b*: when a method is invoked, the program's state is modified by simply adding the effect vector of the call.
3. A *condition vector* noted *b.condition*, randomly set for every branch *b* of a branching block (e.g. Alt): at each branching point, the selected branch is the one whose cosine between its condition vector and the current program state is the greatest.

In order to introduce stochasticity in programs' executions, at each *Call* block traversal a random effect is applied in place of the native one with a probability of *prand*. Native effects are those that have been set during the construction of the structure. In that way, while with $prand = 0$ we simulate a deterministic program, which usually leads to a very restricted view of the program's possibilities, with $prand = 1$ applied effects are different for each execution, and then extracted paths tend to be greatly more diverse.

Algorithm 3 depicts the recursive pseudo-code of the *StateVector* strategy, where *state* stands for the current state vector, *b.effect* is the effect vector associated to the Call block *b*, and *b.condition* is the condition vector array defined for a block *b*. All of these vectors are defined over $[-1;1]^{nbDims}$, where *nbDims* corresponds to the size of the vectors. The *randomVector* function used in Algorithm 3 returns a vector uniformly sampled in $[-1;1]^{nbDims}$.

Four different StateVector strategies, using *prand* set to 0, 0.2, 0.5 and 1, are considered in the following. They are respectively referred to as $StateVector\_0, StateVector\_0.2, StateVector\_0.5$ and $StateVector\_1$. In each, the dimension of the state, effect and condition vectors are set to $nbDims = 20$.

3.3. Metrics and evaluation framework

In past evaluation competitions, whether in grammar inference or in software engineering [34,35], participants were given a set of strings, labeled as positive or negative according to their occurrence in the reference model, and the task to carry out 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.

To evaluate models, negative sequences may be extracted for example in the same manner as in [35] by modifying positive sequences, but from our point of view, evaluating the accuracy of a model by considering that it should or not accept predefined sets of sequences cannot provide fully reliable results. Errors that may occur in a hypothetical model may indeed be so diverse that there usually does not exist any set of negative sequences allowing to well approximate its correctness. Moreover, it is difficult to get enough control on the selection of negative sequences that allow to perform relevant tests of correctness. Rather than considering ratios of string sets that have been correctly classified, we therefore propose to directly perform the evaluation process on both reference and hypothetical FSMs.

Algorithm 3. Function StateVector

```
Input: block, the current block;
Output: seq, a sequence of method labels;
1  $seq \leftarrow \emptyset$; $v \leftarrow 0$;
2  $state \leftarrow initialStateVector$;
3  if $block.type = BlockList$ then
4      foreach b ∈ block.children do $seq \leftarrow seq \cup StateVector(b)$;
5      return seq;
6  end
7  if $block.type = Alt$ then
8      $bmax \leftarrow \mathrm{argmax}_{b \in block.children}\, Cos(state, b.condition)$;
9      return $StateVector(bmax)$;
10 end
11 if $block.type = Loop$ or $block.type = Opt$ then
12     $v \leftarrow Cos(state, block.children[0].condition)$;
13     while $v \geq 0$ do
14         $seq \leftarrow seq \cup StateVector(block.children[0])$;
15         if $block.type = Opt$ then return $seq$;
16         $v \leftarrow Cos(state, block.children[0].condition)$;
17     end
18     return $seq$;
19 end
20 if $block.type = Call$ then
21     Sample $x$ in $[0;1[$;
22     if $x < prand$ then $state \leftarrow state + randomVector()$;
23     else $state \leftarrow state + block.effect$;
24     $sub \leftarrow (block.children.size > 0)\ ?\ StateVector(block.children[0]) : \emptyset$;
25     $seq \leftarrow block.labels[0] \cup sub \cup block.labels[1]$;
26     return $seq$;
27 end
```

While our platform is objective-independent, and that various criteria could be considered with respect to the targeted goal of the inference technique, we choose to focus our experiments on completeness and correctness of the resulting FSM. As done in [36], 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 paths from the hypothetical FSM that belong to the reference one. Given the huge number of possible paths in the FSM (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.

For our experiments we consider subsets of 1000 sequences sampled by the Uniform strategy for both measures. This size appeared to be sufficient to get reliable results in our experiments, since for any of our program configurations and with a maximal trace length of 100 labels, no significant change has been observed by increasing this number.

In the following, we consider the $F1$ score, which is a classical measure allowing to mix recall and precision in a single score: $F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$. Finally, as the size of the output model can be of important matter for readability purposes, we propose to also consider an FS score that includes the number of states of the model: $FS = \frac{F1}{nbStates}$. This normalized score renders the averaged $F1$ contribution per state of a model: it corresponds to a compromise between quality and readability of the produced model.

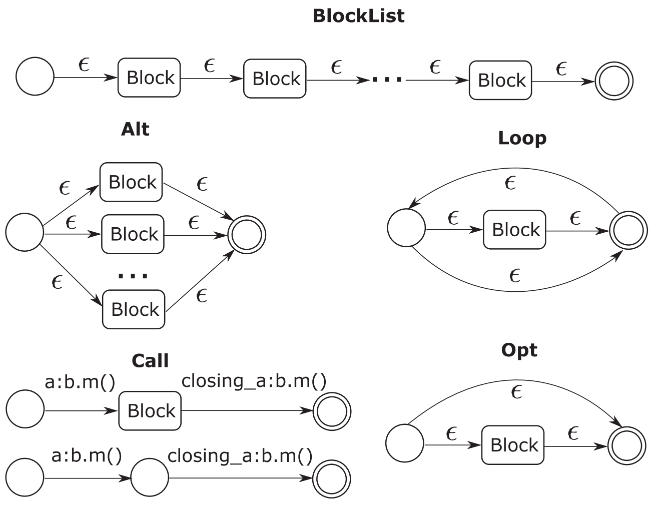

Fig. 7. From blocks to FSM. ϵ symbols stand for empty transitions. The Call blocks use an example of method label with *a* the Caller, *b* the Callee and *m* the name of the method.

4. Experiments using CARE

This section aims at presenting the possibilities of CARE, by detailing some case studies that highlight the benefits resulting from using such an evaluation platform for behavior inference techniques. First we present some experiments on real well-known inference problems and then, we show how an intensive evaluation, thanks to artificially generated data, can serve the analysis of the strengths and the weakness of an inference technique.

While we finally perform a comparison between different techniques, most of our experiments are based on the well-known *KTail* algorithm [8]. The central idea of this algorithm is to consider a state of the model, i.e., the FSM under construction, according to the future behaviors that can been observed from it. After an initialization step in which each trace from the set of observations leads to the production of a specific output branch of the initial state (defining a prefix-tree automaton from traces), *KTail* merges all states that share the same *k* future. Indeed, 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 next *k* labels of paths traversing it. Merging two states then corresponds to assuming a generalization hypothesis: executions sharing the same next *k* labels 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 FSM can be more general with higher values. Thanks to this parameter, *KTail* appears well suited for our experiments, whose primary goal is not to compare state-of-the-art algorithms, but rather to present some use cases of the CARE platform in order to demonstrate its usability and relevance.

4.1. Real case studies

The platform, although designed for intensive evaluation, also allows to manually define programs. This offers the opportunity to evaluate algorithms on real programs, whose behavior model to reach is known, i.e., their reference model. Once the structure is determined, observation traces used may either correspond to artificial traces automatically extracted, or real execution traces the user provides.

To illustrate such an evaluation on real-world programs, we defined the structures of a Concurrent Versions System (CVS), and the X11 Windowing Library, the models of which are given in [16], and of an Automatic Teller Machine (ATM) program whose model is given in [37]. Their respective FSMs are displayed in Appendix A. Table 1 shows results obtained on these three programs by *KTail* with different values of k, in terms of recall (R), precision (P) and number of states of the obtained FSM (S), for two traces extraction strategies that are *Uniform* and *StateVector*_0.2. First, observed results confirm the intuition that low values of k favor completeness while greater values lead to more exact FSM.

On the X11 system, $k = 3$ appears to produce the best model, with perfect recall and precision scores for a smaller FSM than with $k = 5$ and $k = 10$. Since the model of this system is very simple with four main branches and only one loop, the whole behavior of the system is observed, even with a strategy such as *StateVector*_0.2 which usually provides a very restricted view of the various possible paths (see in Section 4.2.2). No differences are then recorded between results from both sets of observations, except for *KTail*_10 showing a very slightly loss in term of recall in the case of *StateVector*_0.2. In that set, no example of loop sequences of 10 statements occur.

ATM is another rather simple system with more nested alternatives but also only one loop. In this case, uniform observations also gather every possible path from the reference model. On that set, results are similar with those observed on the X11 system, except a big loss of precision for KTail_3, whose generalization ability introduced several unobservable paths in the produced FSM. On the other hand, when the view of the whole behavior is more restricted, such as with the *StateVector*_0.2 set of observations, the strong generalization ability of KTail_3 allows it to maintain a perfect recall level, while with greater k values a great loss of recall is recorded.

Finally, on the CVS system, due to the greater complexity of its behavior model with several alternatives, nested loops and optional blocks, extracting a fully complete and correct FSM is a harder task. From the results on this system it appears that, not surprisingly, reaching a good score for recall on the uniform set of observations is easier, but getting a good precision level is harder as more diverse observations lead to more hypothesis on possible paths. On this system, while *KTail*_3 obtains the best recall–precision compromise with observations from *StateVector*_0.2, its dominance is less obvious on the *Uniform* set, i.e. smaller resulting FSM but slightly lower recall/precision levels.

Table 1
Results for KTail on three real programs.

Algo	Prog								
	X11			ATM			CVS		
	R	P	S	R	P	S	R	P	S
Uniform observations									
KTail_1	1.00	0.03	14	1.00	0.06	35	1.00	0.08	33
KTail_3	1.00	1.00	18	1.00	0.25	49	1.00	0.65	75
KTail_5	1.00	1.00	27	1.00	1.00	63	1.00	0.74	135
KTail_10	1.00	1.00	43	1.00	1.00	104	0.99	0.88	387
StateVector_0.2 observations									
KTail_1	1.00	0.03	14	1.00	0.07	35	1.00	0.09	33
KTail_3	1.00	1.00	18	1.00	0.23	53	0.34	0.85	75
KTail_5	1.00	1.00	27	0.27	0.97	71	0.19	1.00	123
KTail_10	0.96	1.00	35	0.24	0.98	101	0.16	1.00	232

When training paths correspond to only a restricted view of the whole behavior, a good generalization ability appears then to be of great importance. A value of $k = 3$ thus seems so far to be the best trade-off, but reliable general conclusions cannot be drawn from such a small and diverse set of tested programs. In order to be able to emit grounded observations, there is a need for conducting experiments on larger sets of data.

4.2. Intensive evaluation

CARE is specifically designed to allow researchers to be able to intensively analyze behavior inference techniques, via the use of artificially generated data. This section first presents properties of these generated data, and then it reports some intensive evaluation experiments, covering core use cases of *CARE*.

4.2.1. Artificial programs

Table 2 reports 7 different distributions of probabilities for block selection $P(t)$. These distributions, used in the experiments that Section 4 reports, imply different levels of difficulty for FSM inference from execution traces. While some configurations lead to structures containing neither alternatives nor loops, others favor complicated structures. Such configurations determine different classes of programs, whose typical corresponding automata are presented in Fig. 8. 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 are not represented in Fig. 8 since they lead to similar automata, with possibly repetitions of program blocks.

Table 3 shows some statistics about the generated programs that we use in our experiments. In this table, and in the remaining of this paper, we consider *maxSizeBlockList* and *maxSizeAlt* (see Section 3.1.2) that limit the maximal number of blocks in a *BlockList* or a *Alt* to 10, and considered a probability for a *Call* to have a nested block *Pchild* = 0.5. Parameter configurations are those that are presented in Table 2, where different probability distributions on the different types of constitutive elements lead to greatly different program structures.

From Table 3, we can indeed note the great impact of the used probability distribution; for instance, some of the configurations lead to structures owning no alternatives (configurations A, B, E and F), others lead to structures with several repetitions of blocks, or to structures with a greater number of Call blocks, etc.

In this table, the number between brackets in each cell stands for the average number of unique occurrences of the type of block in concern for the corresponding program configuration. It highlights the impact of the probability $P(Existing)$ defined in Section 3.1.2. As when this probability is null (configurations A, C and E), no repetitions of block can be observed (each number of unique block equals its corresponding total number of block), when $P(Existing)$ is non null (configurations B, D, F, and G) both numbers

Table 2
Experimented probability distributions.

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

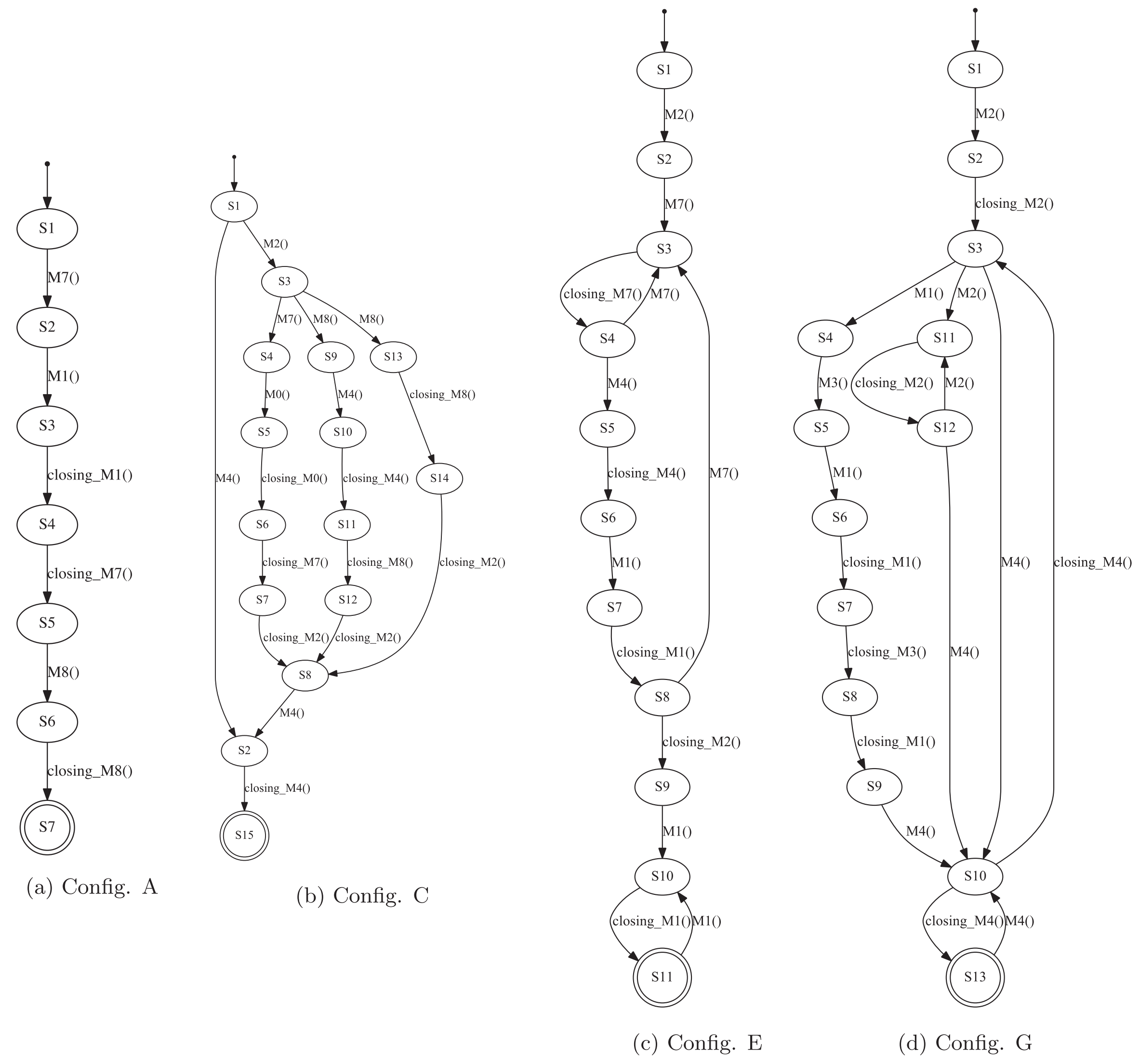

Fig. 8. Tiny examples of typical FSM produced for different configurations of the block selections probabilities. Only configurations A, C, E and G are reported here, others lead to similar FSM with repeated patterns.

Table 3
Average numbers of constitutive block of programs for each configuration from A to G given in Table 2. Each row corresponds to a specific type of block, except the last one which gives global counts of blocks of any type. In each cell, the first number is the average count of the concerned type of block, and the number between brackets corresponds to the number of unique block of this type in the programs (to highlight the possible repetitions of blocks/ labels).

	Element						
Config.	A	B	C	D	E	F	G
BlockList	1.76 (1.76)	1.92 (1.83)	5.39 (5.39)	6.92 (5.15)	1.86 (1.86)	1.94 (1.80)	6.16 (5.21)
Alt	0.00 (0.00)	0.00 (0.00)	5.63 (5.63)	7.54 (4.86)	0.00 (0.00)	0.00 (0.00)	5.93 (4.86)
Opt	0.00 (0.00)	0.00 (0.00)	5.78 (5.78)	9.49 (5.19)	0.00 (0.00)	0.00 (0.00)	6.56 (4.88)
Loop	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	3.45 (3.45)	2.06 (1.53)	6.3 (4.86)
Call	13.72 (13.72)	15.12 (11.49)	50.83 (50.83)	64.48 (35.56)	13.14 (13.14)	15.56 (11.17)	51.08 (37.03)
MethodLabel	13.72 (13.42)	15.12 (11.32)	50.83 (44.85)	64.48 (32.45)	13.14 (12.82)	15.56 (10.95)	51.08 (33.05)
Block	15.48 (15.45)	17.04 (13.32)	67.63 (67.63)	88.43 (50.76)	18.45 (18.45)	19.56 (14.50)	76.03 (56.84)

differ one from the other in each cell, which indicates repeated sub-structures in the generated programs. Note that the *MethodLabel* row is particular since it does not correspond to a type of block but to invocations of methods. As the total number of method invocations always equals the number of call blocks (one invocation per call), the number of unique labels may be greatly lower due to the fact that (1) some labels can belong to repeated call blocks and (2) a same invocation can be sampled multiple times in the program structure. At last, note that configurations with alternatives have significantly greater numbers of blocks, since alternatives imply branching points of the programs with multiple child blocks. Structures from configurations C, D, and G are therefore bigger than others.

4.2.2. Representativeness of the observations

The first experiment reported in this section focuses on the differences between the various traces extraction strategies defined in Section 3.2, in terms of coverage of the whole behavior of the system, which has a great impact on the difficulty of the inference problem. Fig. 9 shows, for different configurations of probability distribution and traces extraction strategies, average scores of recall for 100 programs of each configuration, 100 sequences for each program. These scores correspond to the FSM resulting from the most specific model that only accepts observed sequences, i.e. the prefix tree automaton (precision score is always equal to 1). In all our experiments, we arbitrarily consider 100 as the maximal length of traces drawn from any strategy, and 100 as the size of the observation sets.

With configurations A and B, programs are strictly linear, i.e., they do not contain any branching block. All extracted traces are identical, which results in a recall score equal to 1 whatever the extraction strategy, i.e. the whole behavior has been observed in any case. With other configurations, this global observation is more difficult: branching points of the programs produce many different possible paths. In any of these configurations, we observe that the *prand* variable greatly impacts the representativeness of the set of observations. The lower this stochastic parameter is, the fewer different behaviors are observed, and then the more difficult the problem of modeling the global behavior of the programs.

Unsurprisingly, *Uniform* extraction strategy, that extracts every possible trace with the same probability, forms the most representative set for every program class. These different observation strategies, since leading to different difficulty levels of the modeling problem, greatly serve the full analysis of the algorithms since they offer the opportunity to test their generalization capabilities and robustness.

4.2.3. Example of algorithm analysis

This section aims at giving an idea of which kind of analysis may be made, with artificial programs, using the CARE platform. For this experiment, we focus on the problem of setting an optimal

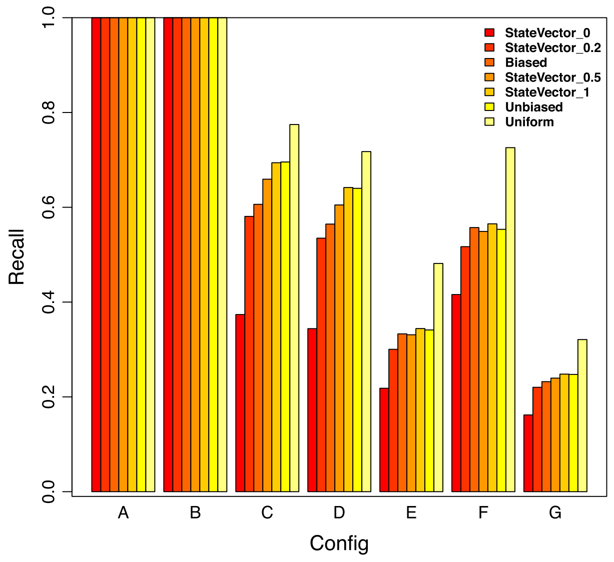

Fig. 9. Representativeness of the observations set for various extraction strategies.

value k for future behaviors considered in *KTail*. We performed an evaluation of *KTail* for different values of k over 1000 artificial programs for each of the seven classes of program defined in Table 2. Each Observations set used by *KTail* contain an arbitrary number of 100 traces extracted following strategies defined in Section 3.2. Tables 4 and 5 give average results obtained with uniform extraction of traces for each class of program.

As expected, while for larger values of k the resulting FSM gets better precision scores (less mergers, the FSM is more specific), with low k recall is favored. When k is set to 0, merges have no constraints, leading to the production of a single-state FSM with as many self transitions as method labels in the set of observations. The rightmost FSM in Fig. 3, representing the most general FSM, illustrates this case. In this case any sequence of observed traces is accepted by the resulting FSM. It thus gets a recall of 1 for each program class with uniform extraction of observations; this is not the case with more restricted views of the program, such as with StateVector_0, for which only a small subset of all method labels of the program is observed. On another hand, it always obtains a Precision score equal to zero since it is very unlikely to sample from such model traces that are contained in the reference. On the other side, k greater than the longest trace from the training set results in an FSA that only accepts observed sequences. *Ktail*_1000 then gets precision scores of 1 for each program class.

Programs from class A and B are linear. In these cases, 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 however contain repetitions of sub-programs, i.e. existing blocks, that may be considered as loops by *KTail* and induce an important loss of precision when mergers are not sufficiently constrained. When considering linear programs, any merger introduces generalization errors.

On the other hand, with programs from class C and E, that respectively contain alternatives or loops, being able to merge states is of great importance in order to detect such branching points. It allows to reduce the size of the produced FSM and to generalize observations into inferred traces that have not been observed but belong to the behavior model of the program. In both cases, $k = 3$ appears to yield the best compromise between completeness and correctness of the produced model, for a reasonably sized FSM.

Programs from classes D and F respectively own similar structures 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 low values of k. Same manner as for class B, repetitions of blocks in the structure introduce important generalization errors. These errors are somewhat balanced by benefits of generalization in presence of alternatives and loops. The same tendencies are observed with programs of class G, that corresponds to the most complex set of programs where every kind of block is possible, and there is a much greater number of possible paths.

Two values for k are highlighted in this experiment. While $k = 5$ allows to limit generalization errors, $k = 3$ appears to obtain rather good accuracy for reasonable sized FSM. However, these results from this easy setting when the uniform strategy is used need to be confirmed with other strategies.

Fig. 10 on the left shows the $F1$ scores obtained with values 1, 3, 5 and 1000 for k, for all program classes and extraction strategies. Differences between scores of these three variants of *KTail* are not constant over the various kinds of sets of observations for each class of program. With 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

Table 4
Results on programs from configurations A, B, C and D for KTail with uniform observations.

Algo	Class											
	A			B			C			D		
	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
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
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
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
Ktail_4	1.00	1.00	33.49	1.00	0.55	30.24	0.98	1.00	106.08	0.97	0.39	106.49
Ktail_5	1.00	1.00	33.49	1.00	0.75	31.07	0.97	1.00	119.75	0.96	0.61	127.30
Ktail_10	1.00	1.00	33.49	1.00	0.92	32.48	0.93	1.00	204.25	0.90	0.82	251.71
Ktail_1000	1.00	1.00	33.49	1.00	1.00	33.32	0.77	1.00	1404.98	0.72	1.00	1680.36

Table 5
Results on programs from configurations E, F and G for KTail with uniform observations.

Algo	Class								
	E			F			G		
	R	P	S	R	P	S	R	P	S
Ktail_0	1.00	0.00	1.00	1.00	0.00	1.00	0.99	0.00	1.00
Ktail_1	1.00	0.73	30.02	1.00	0.33	21.43	0.98	0.30	49.86
Ktail_2	1.00	0.88	36.53	1.00	0.34	25.80	0.97	0.34	73.32
Ktail_3	1.00	1.00	43.23	1.00	0.60	30.02	0.94	0.61	100.04
Ktail_4	0.99	1.00	52.25	1.00	0.60	34.31	0.91	0.62	142.11
Ktail_5	0.99	1.00	62.81	1.00	0.76	39.11	0.85	0.76	200.06
Ktail_10	0.91	1.00	178.47	0.98	0.87	80.56	0.63	0.88	729.88
Ktail_1000	0.48	1.00	3248.20	0.73	1.00	2020.04	0.32	1.00	3877.51

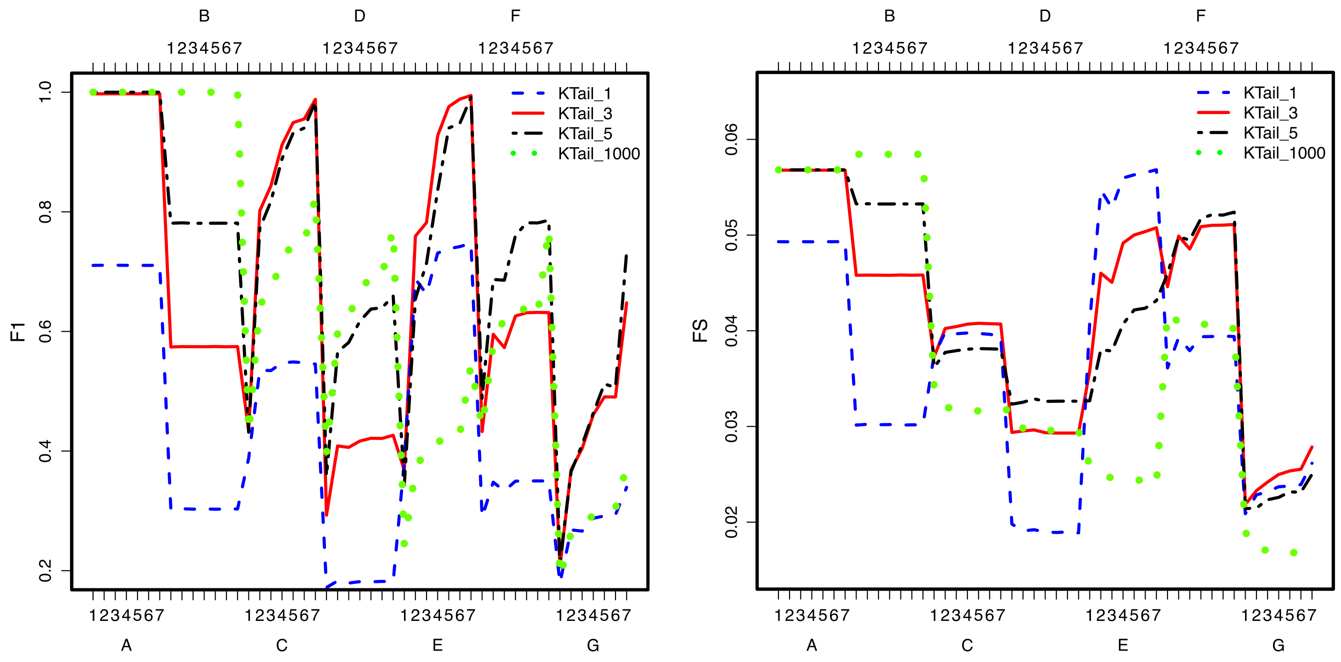

Fig. 10. F1 (on the left) and FS (on the right) scores for *KTail* with *k* set to 1, 3, 5 and 1000. Letters correspond to program classes and numbers to path extraction strategies ordered by increasing coverage of the whole behavior: 1 = *StateVector*_0, 2 = *StateVector*_0.2, 3 = *Biased*, 4 = *StateVector*_0.5, 5 = *StateVector*_1, 6 = *Unbiased*, 7 = *Uniform*.

$k = 3$ is largely greater for restricted views of the whole behavior. See for example class E, where the score difference between $k = 3$ and $k = 5$ is around 0.1 with *StateVector*_0.2, 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 inference techniques.

Fig. 10 on the right, which shows the *FS* scores obtained for the same algorithms, program classes and extraction strategies, confirms the general dominance of *k* values between 3 and 5 for

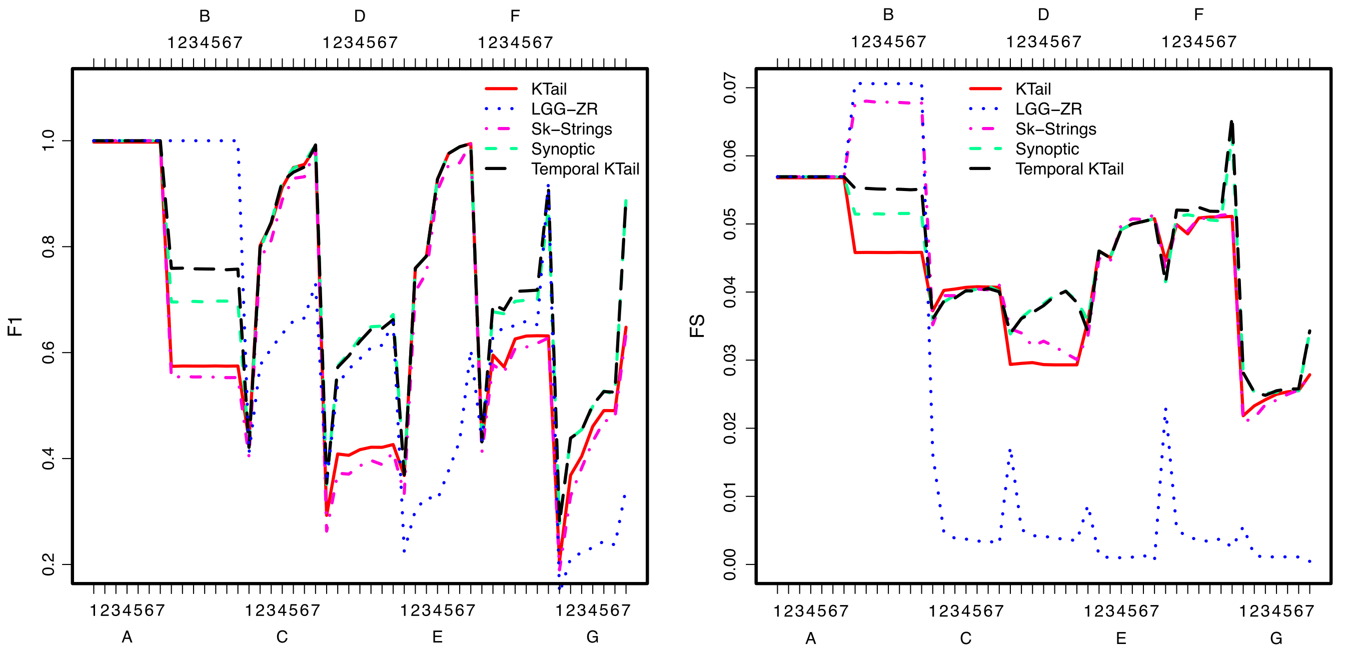

Fig. 11. F1 (on the left) and FS (on the right) scores for KTail, Synoptic, Temporal KTail and LGG-ZR on different problem configurations. Letters correspond to program classes and numbers to path extraction strategies ordered by increasing coverage of the whole behavior: 1 = *StateVector_0*, 2 = *StateVector_0.2*, 3 = *Biased*, 4 = *StateVector_0.5*, 5 = *StateVector_1*, 6 = *Unbiased*, 7 = *Uniform*.

obtaining good compromises between quality and readability of the models. As $k < 3$ usually leads to rather small but too much general models (very low correcteness levels when repetitions of blocks occur such as in configurations B, D, F and G), $k > 5$ tends to produce too large models (beyond sometimes low levels of completeness, the number of states of models obtained with $k = 1000$ is for instance prohibitive). $k = 3$ appears to lead to models with good levels of $F1$ contribution per state in most of configurations.

4.2.4. Example of algorithms comparison

To complete the presentation of the usages of CARE, we propose to compare in terms of $F1$ scores, five different behavior inference techniques (three classical and two state-of-the-art techniques):

- **KTail**: The *KTail* algorithm that has been extensively studied above, with parameter $k = 3$.
- **LGG-ZR**: A classical grammatical inference technique, described in [21], that ensures that the output FSA is a 0-reversible automata that corresponds of the least general generalization of the input traces.
- **Sk-Strings**: An extension of *KTail* for stochastic machines [38]. After having produced an initial automaton (here with *KTail* using $k = 3$), *Sk-Strings* performs merges between states by considering their most probable output sequences: two states x and y can be merged if the most probable k-sized output sequences from x can be accepted by the automaton when starting from y, and reversely. For a same tail size k as for *KTail*, it usually provides more general models (more merges), that ground in the most frequent sequences rather than considering the whole outputs of each state. In the following, we consider results obtained by considering for each state its 50% most probable output sequences of size 4 (longer tails than the *KTail* algorithm used as initial process, but with a more tolerant merge condition).
- **Synoptic**: Considering that long term temporal dependencies may have a great impact on the correctness of a model, Ref. [39] proposed a top-down approach that performs successive refinements of an initial model that violates some temporal constraints. The *Synoptic* approach processes in four steps:
  1. extraction of the temporal dependencies observed in the traces,[2]
  2. initial automaton construction (here we use *KTail* with $k = 3$),
  3. model refinement by following a counterexample guided abstraction approach (CEGAR) [40],
  4. and a final coarsening step to minimize the model.
- **Temporal KTail**: As *Synoptic*, *Temporal KTail* considers long term dependencies extracted from execution traces; these are extracted in the same manner as for *Synoptic* in our experiments. This algorithm, initially proposed in [11] and then revisited in [41] to enhance performances, proposes to follow the same steps as the classical *KTail* algorithm, but while checking before each states merge that this operation would not induce some constraints violations. Here, we consider temporal constraints with atomic pre and post conditions. We also consider that some constraints can be strict (one post-condition occurrence is required per pre-condition occurrence). See [41] for more details.

Fig. 11 shows the $F1$ and FS scores obtained for these four algorithms, for all program classes and extraction strategies, in a similar way as results previously plotted in Fig. 10. Here again, we observe a great effectiveness variability with regards to the considered inference settings. For instance, as $LGG - ZR$ obtains very better results than other techniques on the class B of programs whatever the traces extraction strategy, this dominance is clearly

[2] This is done here by considering a support rate of 20%: to be considered, the pre-condition of the temporal dependency must occur in at least 20% of the input traces, and a confidence rate of 100%: to be considered, the temporal dependency must be observed in every trace where its pre-condition occurs.

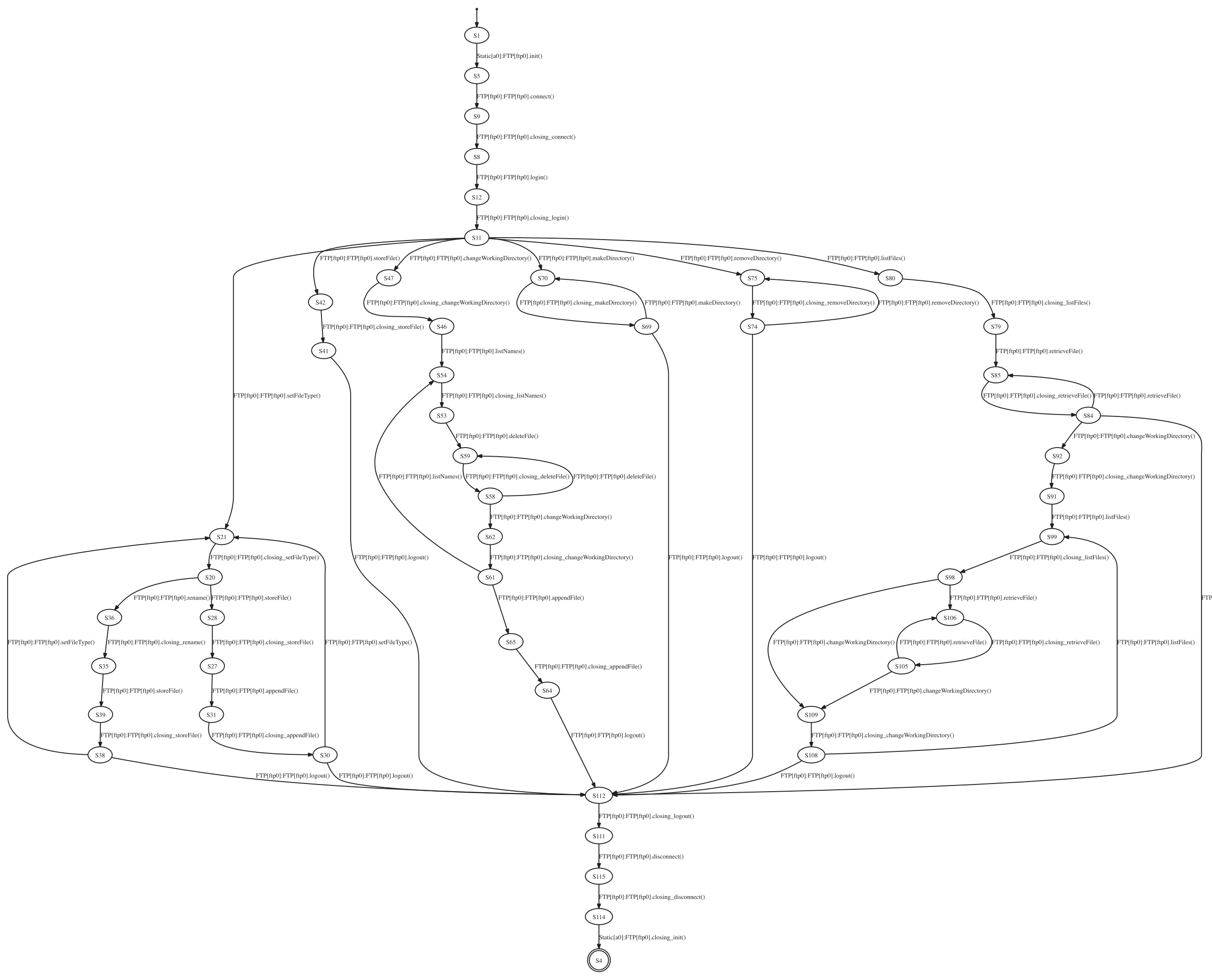

Fig. A.12. Reference FSM of the CVS real case study.

not observed on other programs configurations. It indeed presents great difficulties in detecting alternatives or loops.

We can also note that no general improvements are obtained by *Sk – Strings* compared with *KTail*: A slight lowering of the effectiveness results is observed from *F*1 curves (on the left of the figure), but this is balanced by the tendency of *Sk – Strings* to produce smaller models as observed from *FS* curves (on the right of the figure). We tested various different parameters values for this algorithm, but we never observe better general results than those plotted in Fig. 10. In fact, *Sk – Strings* is not really well fitted for the goal set when considering *F*1 scores, i.e. finding the most complete and exact behavior model of the considered system. It has indeed been initially designed to rather produce probabilistic FSA from traces, and would certainly be more effective for a task of summarizing the observations distributions in a probabilistic automaton (no generalization from the traces). Note that such a task can be considered in CARE by defining adapted evaluations measures, such as the Minimum Message Length measure [42] or Precision/Recall measures with paths samplings respecting probability distributions defined in the models.

At last, considering long term temporal dependencies appears to significantly improve the accuracy of the resulting model. *Synoptic* and *TemporalKTail* indeed clearly obtain the best results in several configurations, notably when the existence of block repetitions leads algorithms as *KTail* to over-generalize. Their relative effectiveness is very close one to the other, but *TemporalKTail* is clearly lighter in term of required resources: in average in our experiments, *Synoptic* required 6 times more processing time than *TemporalKTail* and 2 times more memory.[3]

5. Related work

Three existing investigations [14,17,43] propose an evaluation framework of FSM inference techniques. In this section, we first discuss them. Then, we present other investigations that are related to the contribution presented in this paper.

5.1. Existing evaluation frameworks of FSM inference techniques

The need for a unified framework to compare FSM inference techniques was already identified in the literature [17,35,43]. In this section, we discuss the existing evaluation frameworks.

5.1.1. Walkinshaw et al. [14]

Walkinshaw et al. [14] propose Stamina, as a kind of a competition framework to select the best inference technique. As *CARE*, *Stamina* uses artificial generation to randomly generate a reference FSM on which the inference techniques are evaluated and compared. While *CARE* randomly generates program specifications

[3] *CARE* allows users to easily compare algorithms, as well in term of effectiveness as in term of efficiency, by giving the opportunity of defining several complexity measures.

using probabilities to define the program structure, Stamina extends an existing algorithm initially proposed to generate random complex graphs [44]. The proposed generation procedure is designed to hold some constraints that are present in real programs. However, if the generated graphs are structurally similar to FSM, their similarity with realistic software programs is not ensured. Indeed, complex graphs as presented in [44] are more useful to model networks representing world-wide web and social networks than software programs. As mentioned in this paper, to be more realistic the artificial generation of programs should consider the basic elements representing the structure of programs. Moreover, the *Stamina* framework does not allow to define well-identified classes of program, generated models being only controlled by the number of and out-degree of states to produce.

Following this, the *Stamina* framework generates traces with uniform probability distribution on outgoing transitions, which boils down to our *Unbiased* strategy. As presented in this paper, CARE also proposes three additional strategies for trace extraction.

Finally, we propose the CARE platform not as a competition framework to select the best inference technique. We do not believe that there exists an absolute best inference technique. As shown in our experiments, the same inference technique gives different precision and recall results depending on the class of the used programs. Hence, our goal by proposing the CARE platform is to provide end users with a way to select the inference algorithm that ensures good results according to the class of programs that their program is similar to.

5.1.2. Lo et al. [43]

Lo et al. [43] propose the QUARK framework. Using this framework, reference FSM called simulator models are manually constructed. The particularity of the simulator models is the presence of probabilities on transitions. Indeed, every transition is attached with a probability, indicating how likely the associated method call will be invoked from that source state. As presented above, in the CARE platform we also can add the probability to invoke specific methods in the FSM. However, this information is related to the configuration definition and not to the FSM itself.

In addition, to the manual construction of simulator models, Lo et al. present an algorithm that randomly generates a simulator model. However, these models are only used to test the scalability of FSM inference techniques. In their experiments only the simulator models that are manually constructed are used in the evaluation process.

For trace extraction, the QUARK framework [43] uses a single traversal of the reference FSM model according to the probabilities

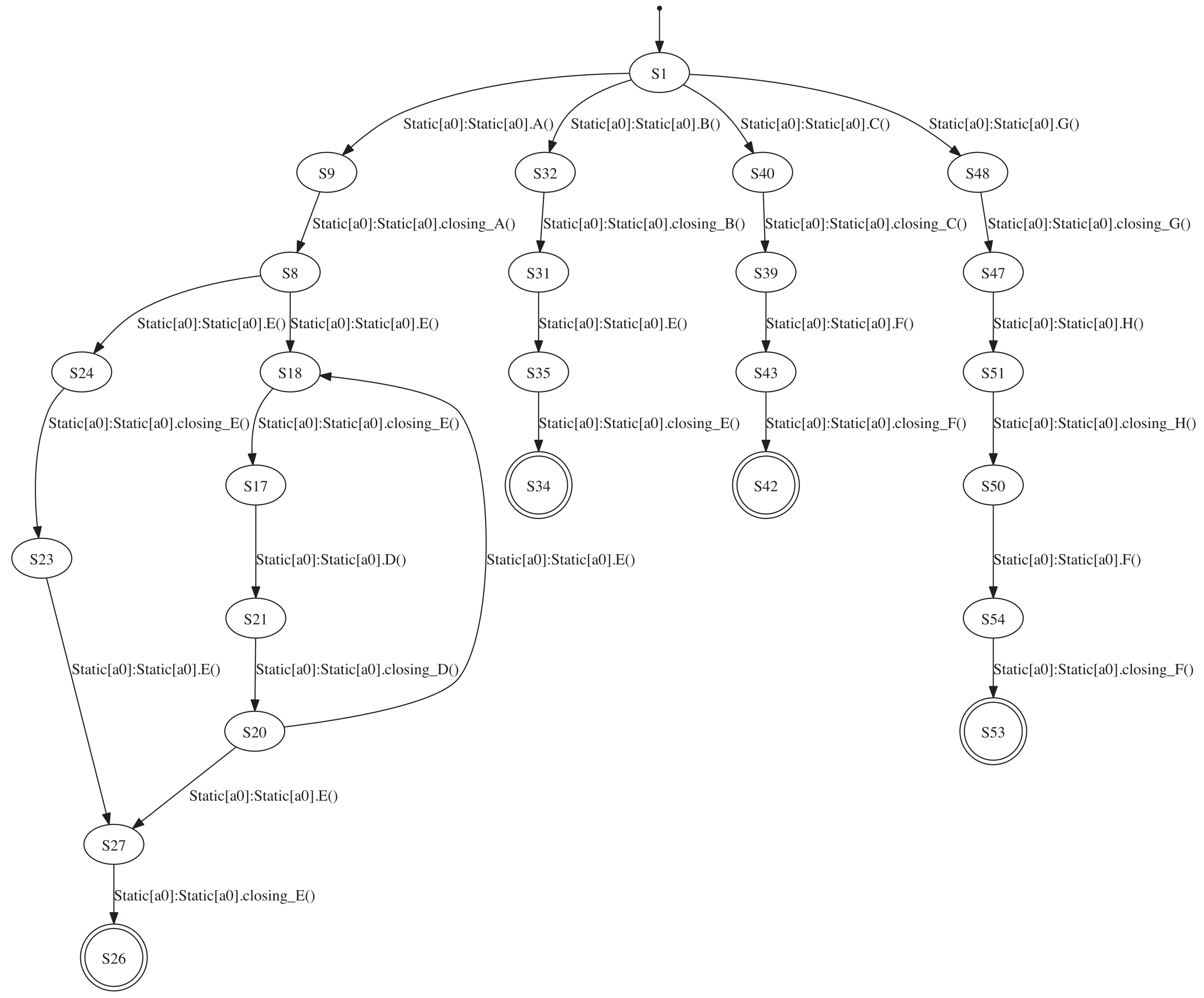
S1
Static[a0]:Static[a0].A()
Static[a0]:Static[a0].B()
Static[a0]:Static[a0].C()
Static[a0]:Static[a0].G()
S9
S32
S40
S48
Static[a0]:Static[a0].closing_A()
Static[a0]:Static[a0].closing_B()
Static[a0]:Static[a0].closing_C()
Static[a0]:Static[a0].closing_G()
S8
S31
S39
S47
Static[a0]:Static[a0].E()
Static[a0]:Static[a0].E()
Static[a0]:Static[a0].E()
Static[a0]:Static[a0].F()
Static[a0]:Static[a0].H()
S24
S18
S35
S43
S51
Static[a0]:Static[a0].closing_E()
Static[a0]:Static[a0].closing_E()
Static[a0]:Static[a0].closing_E()
Static[a0]:Static[a0].closing_F()
Static[a0]:Static[a0].closing_H()
S17
S34
S42
S50
S23
Static[a0]:Static[a0].D()
Static[a0]:Static[a0].E()
Static[a0]:Static[a0].F()
S21
S54
Static[a0]:Static[a0].E()
Static[a0]:Static[a0].closing_D()
Static[a0]:Static[a0].closing_F()
S20
S53
Static[a0]:Static[a0].E()
S27
Static[a0]:Static[a0].closing_E()
S26

Fig. A.13. Reference FSM of the X11 real case study.

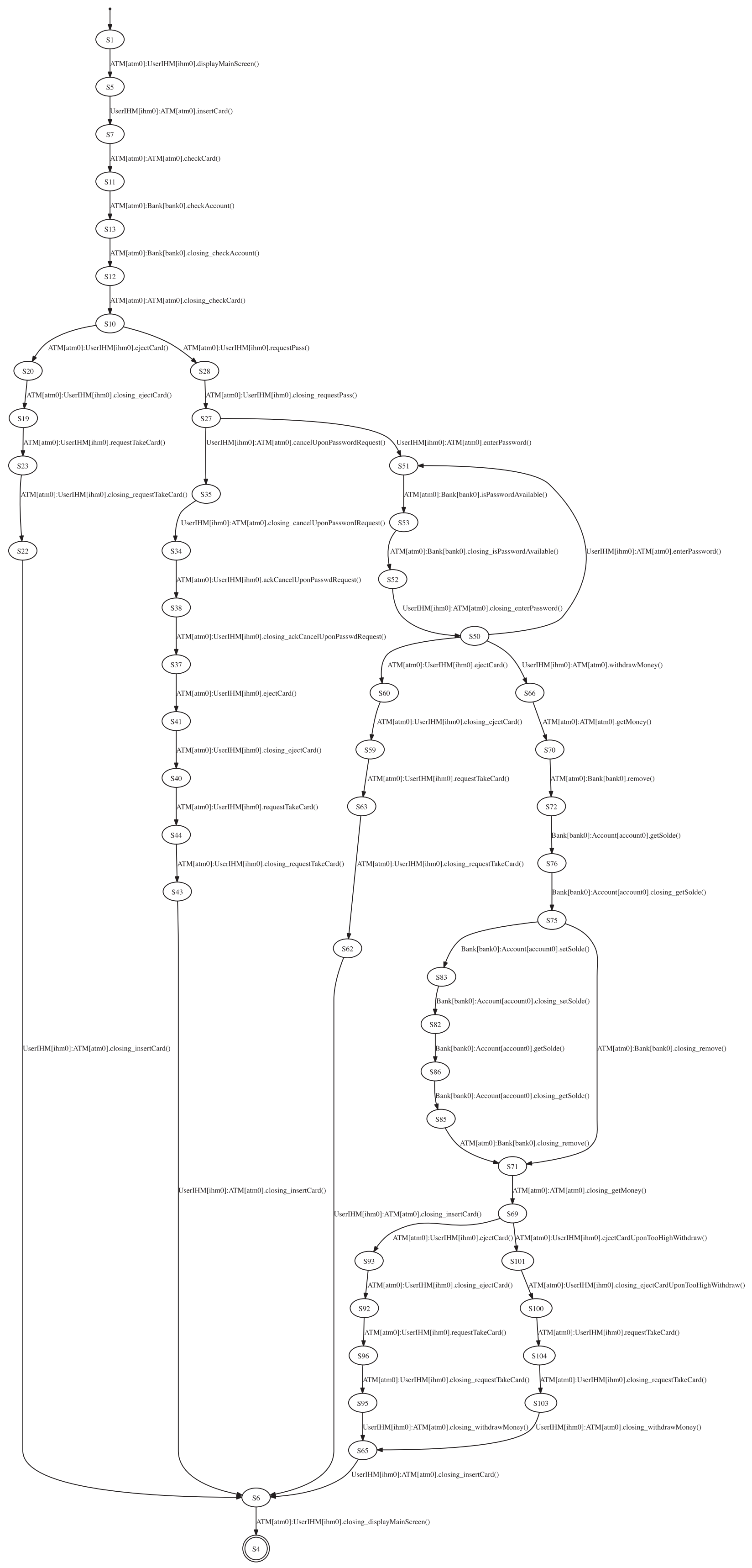
S1
ATM[atm0]:UserIHM[ihm0].displayMainScreen()
S5
UserIHM[ihm0]:ATM[atm0].insertCard()
S7
ATM[atm0]:ATM[atm0].checkCard()
S11
ATM[atm0]:Bank[bank0].checkAccount()
S13
ATM[atm0]:Bank[bank0].closing_checkAccount()
S12
ATM[atm0]:ATM[atm0].closing_checkCard()
S10
ATM[atm0]:UserIHM[ihm0].ejectCard()
ATM[atm0]:UserIHM[ihm0].requestPass()
S20
S28
ATM[atm0]:UserIHM[ihm0].closing_ejectCard()
ATM[atm0]:UserIHM[ihm0].closing_requestPass()
S19
S27
ATM[atm0]:UserIHM[ihm0].requestTakeCard()
UserIHM[ihm0]:ATM[atm0].cancelUponPasswordRequest()
UserIHM[ihm0]:ATM[atm0].enterPassword()
S23
S51
ATM[atm0]:UserIHM[ihm0].closing_requestTakeCard()
S35
ATM[atm0]:Bank[bank0].isPasswordAvailable()
UserIHM[ihm0]:ATM[atm0].closing_cancelUponPasswordRequest()
S53
S22
S34
ATM[atm0]:Bank[bank0].closing_isPasswordAvailable()
UserIHM[ihm0]:ATM[atm0].enterPassword()
ATM[atm0]:UserIHM[ihm0].ackCancelUponPasswdRequest()
S52
S38
UserIHM[ihm0]:ATM[atm0].closing_enterPassword()
ATM[atm0]:UserIHM[ihm0].closing_ackCancelUponPasswdRequest()
S50
S37
ATM[atm0]:UserIHM[ihm0].ejectCard()
UserIHM[ihm0]:ATM[atm0].withdrawMoney()
ATM[atm0]:UserIHM[ihm0].ejectCard()
S60
S66
ATM[atm0]:UserIHM[ihm0].closing_ejectCard()
ATM[atm0]:ATM[atm0].getMoney()
S41
ATM[atm0]:UserIHM[ihm0].closing_ejectCard()
S59
S70
ATM[atm0]:UserIHM[ihm0].requestTakeCard()
ATM[atm0]:Bank[bank0].remove()
S40
ATM[atm0]:UserIHM[ihm0].requestTakeCard()
S63
S72
S44
Bank[bank0]:Account[account0].getSolde()
ATM[atm0]:UserIHM[ihm0].closing_requestTakeCard()
ATM[atm0]:UserIHM[ihm0].closing_requestTakeCard()
S76
S43
Bank[bank0]:Account[account0].closing_getSolde()
S75
S62
Bank[bank0]:Account[account0].setSolde()
S83
Bank[bank0]:Account[account0].closing_setSolde()
S82
UserIHM[ihm0]:ATM[atm0].closing_insertCard()
Bank[bank0]:Account[account0].getSolde()
ATM[atm0]:Bank[bank0].closing_remove()
S86
Bank[bank0]:Account[account0].closing_getSolde()
S85
ATM[atm0]:Bank[bank0].closing_remove()
S71
UserIHM[ihm0]:ATM[atm0].closing_insertCard()
ATM[atm0]:ATM[atm0].closing_getMoney()
UserIHM[ihm0]:ATM[atm0].closing_insertCard()
S69
ATM[atm0]:UserIHM[ihm0].ejectCard()
ATM[atm0]:UserIHM[ihm0].ejectCardUponTooHighWithdraw()
S93
S101
ATM[atm0]:UserIHM[ihm0].closing_ejectCard()
ATM[atm0]:UserIHM[ihm0].closing_ejectCardUponTooHighWithdraw()
S92
S100
ATM[atm0]:UserIHM[ihm0].requestTakeCard()
ATM[atm0]:UserIHM[ihm0].requestTakeCard()
S96
S104
ATM[atm0]:UserIHM[ihm0].closing_requestTakeCard()
ATM[atm0]:UserIHM[ihm0].closing_requestTakeCard()
S95
S103
UserIHM[ihm0]:ATM[atm0].closing_withdrawMoney()
UserIHM[ihm0]:ATM[atm0].closing_withdrawMoney()
S65
UserIHM[ihm0]:ATM[atm0].closing_insertCard()
S6
ATM[atm0]:UserIHM[ihm0].closing_displayMainScreen()
S4

Fig. A.14. Reference FSM of the ATM real case study.

specified on transitions. This strategy is similar to our *Biased* strategy.

5.1.3. Pradel et al. [17]

The evaluation framework of [17] requires human intervention to encode method ordering constraints from which a reference FSM is generated. This framework also defines two novel metrics to compute precision and recall. While the CARE platform formalizes precision and recall in term of inclusion of traces, the Pradel et al.'s framework proposes an algorithm that reuses *KTail* and union between FSM to defines these metrics. As discussed in this paper, one of the objectives of the CARE platform is to be extensible to integrate new evaluation metrics. Hence, we believe that the metrics defined by [17] can easily be integrated in our platform.

5.2. Other related work

As presented in this paper, the CARE platform is based on artificial construction of programs. Construction of artificial benchmarks for building state representations from positive examples has already been investigated by different challenges [34,45]. Nevertheless, most proposals concern the general context of grammatical inference and are not especially adapted to the behavior of software programs [35].

Beschastnikh et al. [46] propose an unified framework to specify the FSM inference algorithms using a declarative specification. This helps to easily analyse and compare the existing algorithms in an unified way. However, this comparison only concerns the algorithmic aspect and not in terms of the quality of the obtained models.

6. Conclusion

By determining structures of programs as nested blocks defining their general behavior, we propose in this paper a way to generate large amounts of 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 and (2) various strategies of execution simulation that lead to different coverage levels of the program behavior.

By experimenting the platform for inference techniques analysis, we observed that inference techniques do not behave in the same manner on every class of program. Being able to characterize different classes of programs is then very useful for understanding the strengths and weaknesses of the studied techniques. On the other hand, experiments also demonstrated the need for inference techniques analysis on various views of the whole program behavior to assess its robustness.

By conducting intensive analysis of a particular inference technique, the well-known *KTail* algorithm, we observed that, while this algorithm performs well for the detection of loops and alternatives, it is greatly less accurate when repetitions of sub-programs occur in the structure of the inferred FSM. Such limitation, that could not be observed with classical experimentation methodologies, may lead to the investigation of an initial step of such repetitions detection to improve inference performance. Moreover, we observed that greater or safer values for its generalization parameter k should be preferred when a good view of the whole behavior of the program is available. This also is an interesting result since it could lead to imagine models for automatic setting of k, according to an inferred representativeness level of the set of observations.

The experiments reported in this paper were designed to demonstrate the core and most important features of the CARE platform. The platform also provides the opportunity to assess the influence of several other parameters on the performances of behavior inference (e.g. number of classes, objects per class, methods per object, maximal depth of the structure, nesting probability, etc.).

Beyond in-depth analysis and comparison of state-of-the-art inference algorithms that the platform allows, it also opens new opportunities for program behavior learning, which was greatly limited by the difficulty of getting sufficiently large and representative sets of training data.

On the platform itself, subsequent work will include the integration of arguments in method invocations during the structure generation in order to be able to consider algorithms such as GKTail [12], that focuses on determining parameter intervals on transitions of the FSM. Considering the goal of behavior inference as producing consistent and readable Sequence Diagrams (SD), we have been investigating new evaluation measures that could consider SD-driven criteria on the resulting FSM, such as divisibility in sub process, coherence of contiguous invocations, one closing per invocation, and consistent order of closings. If such measures turn out to be relevant for evaluation purposes, they also may constitute new criteria to consider in designing inference algorithms.

At last, the CARE platform is initially proposed to generate artificial traces from artificial programs that are used as input to evaluate behavior model inference techniques. Nevertheless, we believe that an interesting perspective can be to explore the use of CARE to evaluate other kind of techniques that are based on programs and traces. For instance, test case generation techniques also need intensive evaluation on artificial programs. However, to support the evaluation of test case generation techniques, we need to give an execution semantics to the generated artificial programs. This perspective requires major modifications to the CARE platform, but it is still a very interesting challenge.

Appendix A. Reference FSM of the CVS, X11, and ATM real case studies

Figs. A.12, A.13, and A.14 depict the reference FSM of the CVS, X11, and ATM real case studies, respectively.

References

[1] D. Harel, E. Gery, Executable object modeling with statecharts, IEEE Comp. 30 (7) (1997) 31–42.

[2] E. Dominguez, B. Pérez, A.L. Rubio, M.A. Zapata, A systematic review of code generation proposals from state machine specifications, Inf. Softw. Technol. 54 (10) (2012) 1045–1066, http://dx.doi.org/10.1016/j.infsof.2012.04.008.

[3] M. Harder, J. Mellen, M.D. Ernst, Improving test suites via operational abstraction, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Comp. Society, Washington, DC, USA, 2003, pp. 60–71. <http://dl.acm.org/citation.cfm?id=776816.776824>.

[4] L. Mariani, S. Papagiannakis, M. Pezze, Compatibility and regression testing of cots-component-based software, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Comp. Society, Washington, DC, USA, 2007, pp. 85–95, http://dx.doi.org/10.1109/ICSE.2007.26.

[5] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263, http://dx.doi.org/10.1145/5397.5399.

[6] S. Sinha, G. Ramalingam, R. Komondoor, Parametric process model inference, in: 14th Working Conference on Reverse Engineering, 2007, WCRE 2007, 2007, pp. 21–30, doi:http://dx.doi.org/10.1109/WCRE.2007.36.

[7] J.E. Cook, A.L. Wolf, Discovering models of software processes from event-based data, ACM Trans. Softw. Eng. Methodol. 7 (3) (1998) 215–249.

[8] A.W. Biermann, J. Feldman, On the synthesis of finite-state machines from samples of their behavior, IEEE Trans. Comp. 21 (1972) 592–597.

[9] S.P. Reiss, M. Renieris, Encoding program executions, in: ICSE 2001, 2001, pp. 221–230.

[10] J. Whaley, M.C. Martin, M.S. Lam, Automatic extraction of object-oriented component interfaces, in: ISSTA 02, 2002, pp. 218–228.

[11] D. Lo, L. Mariani, M. Pezzè, Automatic steering of behavioral model inference, in: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, ACM, New York, NY, USA, 2009, pp. 345–354.

[12] D. Lorenzoli, L. Mariani, M. Pezzè, Automatic generation of software behavioral models, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, New York, NY, USA, 2008, pp. 501–510.

[13] T. Xie, Software Component Protocol Inference, General Examination Report, Univ. of Washington Dep. of Comp. Sc. and Eng., Seattle, WA, June 2003.
[14] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, P. Dupont, STAMINA: a competition to encourage the development and assessment of software model inference techniques, Empir. Softw. Eng. (2012) 1–34, http://dx.doi.org/10.1007/s10664-012-9210-3.
[15] L.C. Briand, Y. Labiche, J. Leduc, Toward the reverse engineering of UML sequence diagrams for distributed java software, IEEE Trans. Softw. Eng. 32 (9) (2006) 642–663.
[16] D. Lo, S.-C. Khoo, SMArTIC: towards building an accurate, robust and scalable specification miner, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, 2006, pp. 265–275.
[17] M. Pradel, P. Bichsel, T.R. Gross, A framework for the evaluation of specification miners based on finite state machines, in: ICSM, IEEE Comp. Society, 2010, pp. 1–10.
[18] S. Lamprier, N. Baskiotis, T. Ziadi, L.M. Hillah, CARE: a platform for reliable comparison and analysis of reverse-engineering techniques, in: ICECCS, IEEE, 2013, pp. 252–255.
[19] P. García, E. Vidal, Inference of k-testable languages in the strict sense and application to syntactic pattern recognition, IEEE Trans. Pattern Anal. Mach. Intell. 12 (9) (1990) 920–925.
[20] D. Angluin, Inference of reversible languages, J. ACM 29 (3) (1982) 741–765.
[21] F. Tantini, A. Terlutte, F. Torre, Sequences classification by least general generalisations, in: Proceedings of the 10th International Colloquium Conference on Grammatical Inference: Theoretical Results and Applications, ICGI'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 189–202.
[22] I. Krka, Y. Brun, D. Popescu, J. Garcia, N. Medvidovic, Using dynamic execution traces and program invariants to enhance behavioral model inference, in: J. Kramer, J. Bishop, P.T. Devanbu, S. Uchitel (Eds.), ICSE, vol. 2, ACM, 2010, pp. 179–182.
[23] T. Arts, S.J. Thompson, From test cases to FSMs: augmented test-driven development and property inference, in: S.L. Fritchie, K.F. Sagonas (Eds.), Erlang Workshop, ACM, 2010, pp. 1–12.
[24] M. Schur, A. Roth, A. Zeller, Mining behavior models from enterprise web applications, in: B. Meyer, L. Baresi, M. Mezini (Eds.), ESEC/SIGSOFT FSE, ACM, 2013, pp. 422–432.
[25] T. Ohmann, K. Thai, I. Beschastnikh, Y. Brun, Mining precise performance-aware behavioral models from existing instrumentation, in: P. Jalote, L.C. Briand, A. van der Hoek (Eds.), ICSE Companion, ACM, 2014, pp. 484–487.
[26] I. Beschastnikh, Y. Brun, M.D. Ernst, A. Krishnamurthy, Inferring models of concurrent systems from logs of their behavior with CSight, in: P. Jalote, L.C. Briand, A. van der Hoek (Eds.), ICSE, ACM, 2014, pp. 468–479.
[27] A. Rozinat, A.K.A. de Medeiros, C.W. Günther, A.J.M.M. Weijters, W.M.P. van der Aalst, The need for a process mining evaluation framework in research and practice, in: A.H.M. ter Hofstede, B. Benatallah, H.-Y. Paik (Eds.), Business Process Management Workshops, Lecture Notes in Computer Science, vol. 4928, Springer, 2007, pp. 84–89.
[28] C. Zhao, K. Ates, J. Kong, K. Zhang, Discovering program's behavioral patterns by inferring graph-grammars from execution traces, in: ICTAI'08, ICTAI '08, IEEE Comp. Society, Washington, DC, USA, 2008, pp. 395–402.
[29] C. Zhao, J. Kong, K. Zhang, Program behavior discovery and verification: a graph grammar approach, IEEE Trans. Softw. Eng. 36 (3) (2010) 431–448.
[30] M. Santoro, Inference of Behavioral Models that Support Program Analysis, Ph.D. thesis, Università degli Studi di Milano-Bicocca, Dottorato di ricerca in INFORMATICA, 23 (08.02.11) <http://hdl.handle.net/10281/19514>.
[31] F.E. Allen, Control flow analysis, SIGPLAN Not. 5 (7) (1970) 1–19, http://dx.doi.org/10.1145/390013.808479.
[32] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, New York, Inc., Secaucus, NJ, USA, 1999.
[33] V. Delmon, Generic epsilon-removal, Tech. rep., Laboratoire de Recherche et Développement de l'Epita, 2007 <https://www.lrde.epita.fr/dload/20070523-Seminar/delmon-eps-removal-vcsn-report.pdf>.
[34] K.J. Lang, B.A. Pearlmutter, R.A. Price, Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm, in: ICGI'98, 1998, pp. 1–12.
[35] N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, P. Dupont, A framework for the competitive evaluation of model inference techniques, in: Proceedings of the First International Workshop on Model Inference in Testing, MIIT '10, ACM, New York, NY, USA, 2010, pp. 1–9.
[36] D. Lo, S.-C. Khoo, QUARK: empirical assessment of automaton-based specification miners, in: WCRE '06, IEEE CS, Washington, DC, USA, 2006, pp. 51–60.
[37] S. Uchitel, J. Kramer, J. Magee, Synthesis of behavioral models from scenarios, IEEE Trans. Softw. Eng. 29 (2) (2003) 99–115.
[38] A. Raman, J. Patrick, P. North, The SK-strings method for inferring PFSA, in: Proceedings of the Workshop on Automata Induction, Grammatical Inference and Language Acquisition, 1997, 1997.
[39] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, M.D. Ernst, Leveraging existing instrumentation to automatically infer invariant-constrained models, in: T. Gyimóthy, A. Zeller (Eds.), SIGSOFT FSE, ACM, 2011, pp. 267–277.
[40] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: E. Emerson, A. Sistla (Eds.), Computer Aided Verification, Lecture Notes in Computer Science, vol. 1855, Springer, Berlin Heidelberg, 2000, pp. 154–169.
[41] S. Lamprier, T. Ziadi, N. Baskiotis, L.M. Hillah, Exact and efficient temporal steering of software behavioral model inference, in: ICECCS'14, IEEE Comp. Society, Tianjin, China, 2014, pp. 166–175.
[42] M.P. Georgeff, C.S. Wallace, A general selection criterion for inductive inference, in: ECAI, 1984, pp. 219–228.
[43] D. Lo, S.-C. Khoo, QUARK: empirical assessment of automaton-based specification miners, in: IEEE Comp. Society, Springer, 2006, pp. 51–60.
[44] J. Leskovec, J.M. Kleinberg, C. Faloutsos, Graph evolution: densification and shrinking diameters, in: TKDD, vol. 1(1).
[45] S. Verwer, R. Eyraud, C. de la Higuera, Results of the PAutomaC probabilistic automaton learning competition, J. Mach. Learn. Res. – Proc. Track 21 (2012) 243–248.
[46] I. Beschastnikh, Y. Brun, J. Abrahamson, M.D. Ernst, A. Krishnamurthy, Unifying FSM-inference algorithms through declarative specification, in: D. Notkin, B.H.C. Cheng, K. Pohl (Eds.), ICSE, IEEE/ACM, 2013, pp. 252–261. <http://dl.acm.org/citation.cfm?id=2486788>.