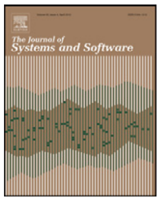
Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

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



Learning extended FSA from software: An empirical assessment

David Lo[a], Leonardo Mariani[b,*], Mauro Santoro[b]

[a] School of Information Systems, Singapore Management University, Singapore
[b] Department of Informatics, Systems and Communication, University of Milano Bicocca, Viale Sarca, 336 Milano, Italy

ARTICLE INFO

Article history:
Received 29 July 2011
Received in revised form 25 March 2012
Accepted 1 April 2012
Available online 25 April 2012

Keywords:
FSA inference
Empirical validation
Behavioral models

ABSTRACT

A number of techniques that infer finite state automata from execution traces have been used to support test and analysis activities. Some of these techniques can produce automata that integrate information about the data-flow, that is, they also represent how data values affect the operations executed by programs.

The integration of information about operation sequences and data values into a unique model is indeed conceptually useful to accurately represent the behavior of a program. However, it is still unclear whether handling heterogeneous types of information, such as operation sequences and data values, necessarily produces higher quality models or not.

In this paper, we present an empirical comparative study between techniques that infer simple automata and techniques that infer automata extended with information about data-flow. We investigate the effectiveness of these techniques when applied to traces with different levels of sparseness, produced by different software systems. To the best of our knowledge this is the first work that quantifies both the effect of adding data-flow information within automata and the effectiveness of the techniques when varying sparseness of traces.

1. Introduction

In the recent years, several techniques that automatically derive behavioral models from execution traces have been used to support validation and verification activities (Lorenzoli et al., 2008; Dallmeier et al., 2006; Ernst et al., 2001; Hangal and Lam, 2002; Henkel and Diwan, 2003; Mariani and Pastore, 2008; Raz et al., 2002; Ammons et al., 2002). A kind of model that is both commonly used to represent program behaviors and largely supported by model generation techniques is Finite State Automaton (FSA) (Lorenzoli et al., 2008; Dallmeier et al., 2006; Mariani and Pastore, 2008; Ammons et al., 2002; Biermann and Feldman, 1972).

Recent results highlighted that many program behaviors cannot be accurately represented with FSAs, but require models that include not only the information about the possible sequences of events executed by programs but also data-flow information (Lorenzoli et al., 2008; Mariani and Pastore, 2008; Mariani et al., 2011a). For instance, the parameters that a program uses to open a file can influence the way the file will be successively used (e.g., opening a file with the read option does not allow values to be written into the file). Simple FSAs can represent the different ways a file can be used (e.g., reading or writing), but cannot relate the specific usage to the opening mode.

For instance gkTail (Lorenzoli et al., 2008) and KLFA (Mariani and Pastore, 2008) are two techniques that can extract FSAs that incorporate data-flow information. In particular, gkTail builds FSAs where transitions are annotated with data-flow information consisting of algebraic constraints. The algebraic constraints associated with transitions specify the concrete values that can be assigned to attributes, thus representing with a single transition a large (possibly infinite) set of events. Fig. 1 shows an example EFSA[1] inferred by gkTail. KLFA builds simple FSAs with transition labels that incorporate data-flow information consisting of universally quantified constraints. Universally quantified constraints specify how data values can reoccur across events. Also in this case the inferred FSA can represent a (possibly infinite) set of events with a single transition. Fig. 2 shows an example EFSA inferred by KLFA.

The models generated by KLFA and gkTail can effectively represent data-flow information through algebraic and universally quantified constraints, compared to simple FSAs. However it is still unclear whether automatically inferred models extended with data-flow information are generally more accurate than simple

* Corresponding author.
E-mail address: mariani@disco.unimib.it (L. Mariani).

[1] In the rest of the paper we will interchangeably use the terms extended FSA (EFSA) and FSA with data-flow information to indicate models extended with either algebraic or universally quantified constraints.

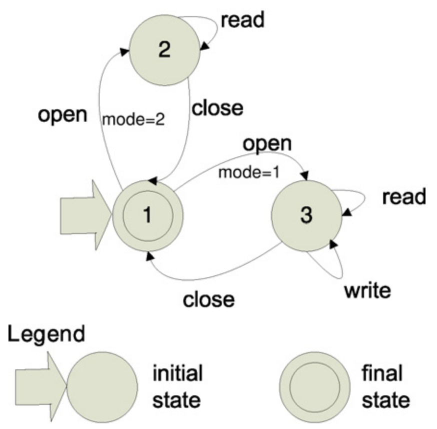


Fig. 1. An extended FSA that describes how a file is used taking into account how the file is opened (mode = 1 corresponds to read/write access, mode = 2 corresponds to read access). This kind of FSA can be inferred by gkTail.

FSAs or not. In particular, processing multiple types of information (event sequences and data-flow) is harder than processing event sequences only, and the quality of the resulting model can be compromised by the complexity of the inference process. In addition, processing more data means spending more time for the inference. In some cases, the time necessary to produce the models can be extremely important.

In this paper, we present an *empirical comparative study* between techniques that infer FSAs and techniques that infer EFSAs. The study focuses on *event traces*, that is traces with event names annotated with attribute values (e.g., sequences of method invocations annotated with parameter values), collected with passive methods (i.e., we simply monitor and record event traces). Event traces can be easily collected from any software system by monitoring interactions between components without requiring access to the component state (black-box monitoring). We do not consider inference of FSAs from *state traces*, that is traces with event names and state values (e.g., sequences of method invocations annotated with the values of the state variables of the objects that process the invocations), because their applicability is restricted by the need of a strategy for recording the value of state variables (white-box monitoring).

The objective of this comparison is to study the tradeoff between inferring FSAs and EFSAs, with *specific reference to models extracted from software systems*. The empirical study evaluates the effectiveness of these techniques while varying the set of available traces from *sparse*, which is an extremely common case when traces are collected by testing software systems, to *dense*, which only happens for thoroughly tested systems. We compare techniques that infer FSAs and EFSAs according to the quality of the inferred models and the time required to obtain them. We conclude our study providing remarks about how to choose a proper model generation technique according to the kind of analysis it should support and the characteristics of the available traces.

According to our investigation of the state of the art, the only inference techniques that can generate FSAs annotated with algebraic and universally quantified constraints from event traces, and have been used to build behavioral models from real software systems, are KLFA (Mariani and Pastore, 2008) and gkTail (Lorenzoli et al., 2008). Since KLFA and gkTail are obtained by adding a constraint identification stage to kBehavior (Mariani et al., 2011b) and kTail (Biermann and Feldman, 1972), which are algorithms that can infer simple FSAs, this empirical study compares the performance of these four techniques on a same set of case studies. In this way, differences on both the quality of the models and the performance strictly result from the inference of data-flow information within the models. These differences carry insights that software engineers can use to decide, case by case, if data-flow information should be considered or not. Since the study focuses on constraints, we do not consider techniques that can decorate models with annotations different from constraints, such as probabilities (Ammons et al., 2002; Cook and Wolf, 1998), and techniques that can infer simple FSAs with a process that is unrelated with the processes used by KLFA and gkTail, e.g., approaches that actively issue queries to end users or generate new test cases (Raffelt and Steffen, 2006; Walkinshaw et al., 2007; Bertolino et al., 2009).

To the best of our knowledge, this paper is the first empirical study that (1) compares and quantifies the tradeoffs between the techniques that infer simple FSAs and the techniques that incorporate data-flow information within FSAs; (2) measures the impact of the sparseness of the traces on the model; and (3) compares a core set of techniques using 10 models extracted from third-party software systems. Most of previous studies in inference of behavioral models from software systems validated the techniques with fewer cases without always including direct comparison with similar techniques (Ramanathan et al., 2007; Walkinshaw and Bogdanov, 2008; Lo and Maoz, 2009; Thummalapenta and Xie, 2009; Zhong et al., 2009).

The main findings reported in this paper are:

- the recall and precision of the models decrease if data-flow information is incorporated in the model: the loss depends from the kind of information that is added to the model and the sparseness of the traces;
  - in the empirical validation the maximum loss of recall has been 0.2 when adding algebraic constraints and 0.54 when adding universally quantified constraints;
  - in the empirical validation the maximum loss of precision has been 0.33 when adding algebraic constraints and 0.64 when adding universally quantified constraints;
- good simple FSAs can be inferred even from sparse sets of executions: according to our experiments a set of executions that covers all the states in the model to be inferred, but not all transitions, is sufficient to generate FSAs with good precision and recall;
- good FSAs with algebraic or universally quantified constraints require a dense set of executions to be inferred: according to our experiments a set of executions that covers all the transitions in the model to be inferred is usually needed;
- the inference of FSAs with algebraic constraints is multiple orders of magnitudes slower than the inference of FSAs with

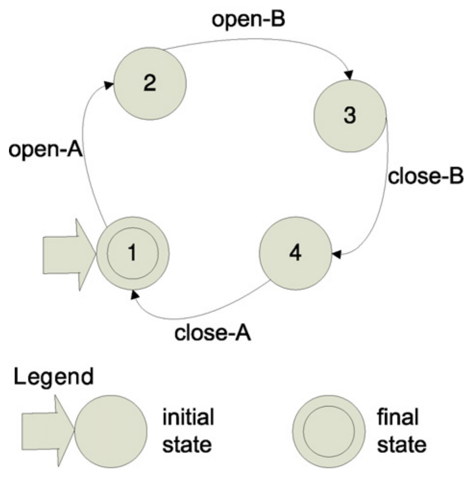


Fig. 2. An EFSA that describes how two files are sequentially opened and closed. This kind of FSA can be inferred by KLFA.

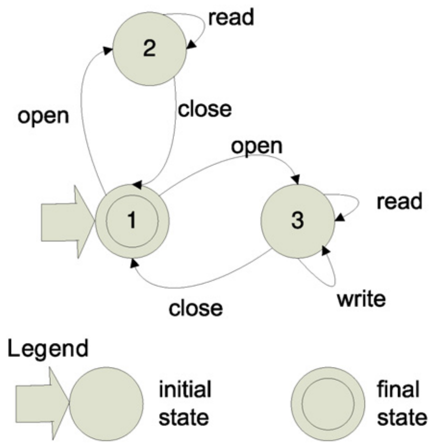


Fig. 3. A simple FSA that describes the usage of a file. This kind of FSA can be inferred by both kTail and kBehavior.

universally quantified constraints or simple FSAs: on average inference with gkTail required 38 min compared to few seconds required by KLFA, kTail and kBehavior.

The paper is organized as follows. Section 2 describes the goals of the empirical validation presented in this paper and defines the main research questions that are investigated. Section 3 shortly describes the techniques that are compared in the empirical study. Section 4 presents the empirical process that we follow to answer each research question. Section 5 describes the toolset used to run the empirical validation. Section 6 presents and discusses the empirical data. Section 7 summarizes the main findings of our empirical work. Section 8 discusses the threats to the validity of the empirical validation. Section 9 discusses related work. Finally, Section 10 summarizes the contributions of this paper.

2. Goals of the assessment

Software systems have more complex behaviors than those that can be represented with simple FSAs. For instance, the simple FSA in Fig. 3 represents how an application uses a file. The real usage of the file is abstracted in several ways. The choice between reading only, or reading and writing values, is presented as a non-deterministic choice, while in reality it is determined by the opening mode. This lack of information can result in an imprecise model-based analysis. For instance, such FSA, when used for analysis, cannot detect that an application opens a file with read mode and then illegally attempts to write values into the file.

Simple FSAs can be extended with constraints that can compactly represent behaviors like the one exemplified above. For example, a more detailed model for the file reading/writing scenario is shown in Fig. 1. Although this extension is indeed useful from a conceptual point of view, EFSAs are more challenging to be inferred than simple FSAs. The attempt to learn an EFSA could end up with an imprecise model that improperly represents both complex and simple behaviors. On the contrary, the inference of an FSA, even if not including any data-flow information, could end up with a model that precisely represents at least the simple behaviors. In the literature, there exists no comparative and quantitative empirical study that confirms or rejects the hypothesis that more accurate models can be learnt by using techniques that learn EFSAs than those that learn FSAs. The goal of our empirical study is to quantify and compare the performance of techniques that infer simple FSAs with techniques that infer EFSAs.

Our empirical comparison investigates the following three research questions:

R1 Do inference techniques producing extended FSAs generate models that can better identify legal behaviors as compared to those producing simple FSAs?
R2 Do inference techniques producing extended FSAs generate models that can better reject illegal behaviors as compared to those producing simple FSAs?
R3 What is the performance difference between the generation and the checking of extended FSAs as compared to those of simple FSAs?

Note that these research questions are extremely important when these models are used to support automated software analysis, but they are not necessarily critical when models are used for other purposes. For instance if the models must be manually inspected, human readability can be more relevant than model accuracy. We refer interested readers to Cornelissen et al. (2009) for a survey of dynamic analysis techniques that can be used to support program comprehension.

3. FSA inference techniques

In this section, we shortly present the inference techniques compared in our empirical evaluation.

3.1. kTail

kTail is a technique that generates a FSA from a set of traces in two steps (Biermann and Feldman, 1972). In the first step it builds a Prefix Tree Acceptor (PTA), which is a tree where edges are labeled with event names. The language accepted by the PTA exactly consists of the set of event sequences recorded in the traces. In the second step kTail transforms, through a heuristic, the PTA into a FSA. The heuristic consists of merging two states if they share the same future of length *k*. The future of length *k* of a state is defined as the set of the event sequences of maximum length *k* that can be accepted from the state. The final automaton is obtained by merging every pair of states with the same future of length *k*.

Fig. 3 shows a sample automaton that can be inferred by kTail.

3.2. kBehavior

kBehavior is a technique that incrementally generates a FSA from a set of traces (Mariani et al., 2011b). When a new trace is submitted to kBehavior, kBehavior first identifies sub-traces of the input trace that are accepted by sub-automata in the current automaton (the sub-traces must have a minimal length *k*, otherwise they are considered too short to be relevant). Then kBehavior extends the automaton with the addition of new branches that suitably connect the identified sub-automata, producing a new version of the automaton that accepts the entire input trace. In complex cases, kBehavior can work recursively to extend a model with newly generated automata rather than branches. We refer interested readers to Mariani et al. (2011b) for a detailed description.

Fig. 3 shows a sample automaton that can be inferred by kBehavior.

3.3. gkTail

gkTail is a technique that generates an EFSA from a set of traces that incorporate information about both the event sequences and the values of the parameters associated with event sequences (Lorenzoli et al., 2008). The technique can be seen as an extension of kTail to produce FSAs with transitions annotated by algebraic constraints.

gkTail works in four steps. In the first step, gkTail merges similar traces, that is traces that share the same event sequence but differ on the values assigned to parameters. Merging a set of similar traces produces a single trace where each event is annotated with the set of values obtained as the union of every value assigned to this same event in every merged trace. In the second step, gkTail mines an algebraic constraint from the values associated with each event. In the third steps, gkTail uses the event sequences annotated with algebraic constraints to build a PTA where transitions are annotated with constraints. In the last step, the states with the same future of length k are merged.

In the case of EFSAs, the future of a state consists of a set of event sequences annotated with algebraic constraints. The future of two states is equivalent if the states accept the same sequences of events and the events in the future are annotated with equivalent constraints, that is the first constraint logically implies the second, and vice versa. The final EFSA is obtained after merging every pair of states with the same future. Fig. 1 shows a sample automaton that can be inferred by gkTail.

3.4. KLFA

KLFA is a technique that generates an FSA from a set of traces that incorporate information about both the event sequences and the values of the attributes associated with events (Mariani and Pastore, 2008). The generated FSA has special transition labels that include data-flow symbols (e.g., A and B in Fig. 2) in addition to the event name. A data-flow symbol is a string that encodes a universally quantified constraint. Universally quantified constraints represent how attribute values reoccur across events.

KLFA works in two steps. In the first step, KLFA analyzes the traces to discover universally quantified constraints. KLFA encodes these constraints in the event names by replacing concrete values with symbols that represent the discovered recurrence patterns. On a syntactical point of view, the rewritten traces are pure event sequences. In the second step, KLFA uses kBehavior to generate the FSA from the rewritten traces.

KLFA implements multiple rewriting strategies that can handle different types of patterns. For example, the global ordering rewriting strategy consistently replaces the same concrete value with a same symbol. In this way, the model is independent from the concrete values that occur in the traces (concrete values are replaced with symbols), but still captures how the same value re-occurs among events (this case is captured by the same symbol repeated among multiple events). For instance, the symbol A associated with events open and close in the FSA shown in Fig. 2 indicates that the same file is opened and then closed.

To properly use symbols, KLFA heuristically separates parameter values into multiple groups of homogeneous values (e.g., a group for values in meters and a different group for values in Kelvin) and then rewrites the values in each group independently from the other groups. If too many symbols are necessary to rewrite the values in one group, KLFA heuristically assumes that a proliferation of symbols is the consequence of the lack of patterns that can be discovered. In that case, KLFA drops the values in the group and uses events without data-flow information for the inference of the model.

In this paper we use the global ordering rewriting strategy because it is the strategy used when no information about the nature of the traces is available. See Mariani and Pastore (2008) for details about the other rewriting strategies.

Fig. 2 shows a sample automaton that can be inferred by KLFA.

4. Empirical setup

In the empirical assessment, kTail, kBehavior, gkTail and KLFA are used to learn 10 models extracted from 10 third-party software systems: Jfreechart (2011), Lucane (2011), Thingamablog (2011), Jeti (2011), Columba (2011), Open Hospital (2011), Rapid Miner (2011), Heritrix (2011), Jargs (2011) and TagSoup (2011).

We selected the models according to the following process. We browsed the Internet for open-source software developed by independent parties. We focused on open-source software because we need to manually derive the ideal models that should be learned by the techniques from the source code. We found the 10 applications considered in this study by browsing SourceForge (2011) and the related web sites looking for applications from different domains. This heterogeneity is useful to avoid selecting multiple applications from the same domain, which may bias the study. In each application, we concentrated on models that represent the method invocations that can be generated when executing a method of the program. We choose such scale of models, opposed to models that represent the entire execution flow of a program, for two reasons: (1) many testing and analysis techniques use model inference to produce models of that scale (Dallmeier et al., 2010; Ammons et al., 2002; Reiss and Renieris, 2001; Lorenzoli et al., 2008), thus it is considered an important scale by the scientific community; (2) it is feasible to manually produce the ideal models from the source code. A representation of the ideal models that must be learnt is necessary to measure the quality of the inferred models. Considering huge models that represent the entire behavior of an application would make the manual derivation of the ideal models very hard or even practically infeasible. Finally, to avoid that a particular programming style adopted by an application could bias the results of our study, we designed the study to cover as many diverse applications as possible. In this way the impact of any factor specific to a single application on the study is minimized. In the practice we selected one method per application resulting in 10 methods extracted from 10 applications.

We identified the methods for our empirical study by looking at methods with behaviors of increasing complexity. We evaluated the complexity of a method by manually deriving the corresponding model (that should be learned by techniques) and then measuring its cyclomatic complexity (CC). CC indicates how complex the structure of the graph (hence the FSA) is. To avoid studying trivial cases, we selected methods with $CC > 5$, and we classified the methods, and hence the FSAs to be learned, in three classes: $CC < 10$, $10 \leq CC \leq 15$ and $CC > 15$. Since we want to evaluate and compare the effectiveness of techniques for inferring FSAs and EFSAs, we also required that the behavior of the selected methods includes exchanging parameters with other methods. Columns *Application* and *Method name* in Table 1 indicate the applications and the methods used in the empirical study, respectively. Column CC specifies the cyclomatic complexity of the behavior produced by each method considered in the empirical evaluation.

To evaluate the quality of an inferred model, we need an ideal model that represents the behavior of the method under analysis. To produce such a model, we analyzed the source code, manually produced the EFSA that matches the behavior of the program, and executed test cases to check the correctness and verify the feasibility of the behaviors represented in the model.

We represented each ideal model extracted from the source code as an EFSA with the following semantics:

- *transitions labels* are method signatures and represent method invocations.
- *transition constraints* are Boolean expressions that represent constraints on the values that can be assigned to program variables and parameters. For example, a constraint `file.status == 0` associated with the transition label `open(file, mode)` indicates that only a closed file is accepted as a parameter.
- *parameter names* have global semantics, that is if the same variable name reoccurs in different signatures and constraints, its

Table 1
Data about the reference EFSAs.

Application	CC		Num. states	Num. transitions	Num. constraints	Method name
	Class	Value				
Lucane	CC < 10	7	11	16	26	MessageHandler.run
Columba		7	13	18	14	FetchNewMessagesCommand.execute
Jeti		8	12	18	38	Jeti.actionPerformed
ThingamaBlog	10 ≤ CC ≤ 15	10	10	18	18	ParagraphComboHandler.actionPerformed
Open Hospital		11	23	32	23	PatientBillEdit.getJButtonSave
Rapid Miner		15	19	32	16	DBScan.generateClusterModel
Heritrix		15	19	32	23	ExtractorHttp.innerProcess
JArgs		15	15	28	28	CmdLineParser.parse
JFreeChart	CC > 15	17	11	26	38	ChartFactory.createPieChart
Tagsoup		27	10	35	14	HTMLScanner.scan

Column application indicates the name of the application that contains the analyzed method. Columns CC, Num. states, Num. transitions and Num. constraints indicate the cyclomatic complexity, the number of states, the number of transitions and the number of constraints in the ideal model extracted from the method under analysis, respectively. Column method name specifies the analyzed method.

value in an execution must always be the same. This semantics allows the generation of traces that are coherent with the behavior of the programs. If a parameter with the same name must be assigned with different values in different transitions, we simply change the name of the parameter in the different transitions to preserve the global semantics.

For the rest of the paper, we refer to the ideal EFSAs as the *reference EFSAs*. Fig. 4 shows the reference EFSA for the Lucane application, which is one of the case studies considered in this paper. Table 2 specifies identifiers of labels and constraints associated with each transition. Table 3 specifies the actual values of labels and constraints for each identifier. Detailed data about the EFSAs used for the empirical assessment is presented in Table 1: Columns *Num. states*, *Num. transitions* and *Num. constraints* specify the number of states, transitions and constraints in the reference model, respectively.

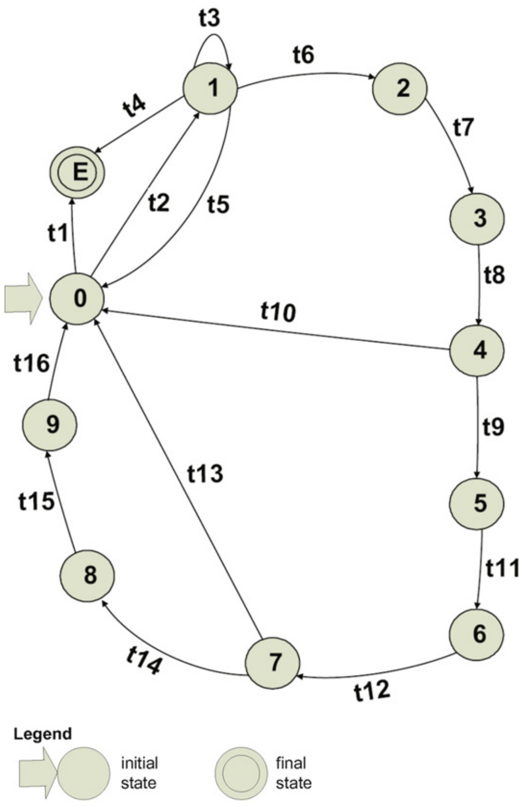


Fig. 4. The reference EFSAs extracted from Lucane.

Table 2
Mapping from transition to method and constraint ids.

Edge	Method	Constraints
t1	m1	{}
t2	m1	{}
t3	m1	{c1, c2, c3}
t4	m1	{c1, c2, c3}
t5	m2	{c4}
t6	m3	{!c1, !c2, !c3, !c4}
t7	m4	{c5}
t8	m5	{c6}
t9	m6	{c7}
t10	m5	{!c7}
t11	m7	{c5, c7, c8}
t12	m8	{c6, c9}
t13	m9	{c8, c9, !c10}
t14	m9	{c8, c9, c10}
t15	m10	{}
t16	m11	{}

The effectiveness of the techniques experienced in our evaluation is influenced by two main factors: the value of the parameter k, which determines how much the inference techniques generalize the behavior represented by traces, and the completeness of the set of traces used to infer the models. In this study, the primary goal is to evaluate the effectiveness of the inference techniques when varying the completeness of the available traces. Intuitively, it corresponds to measuring the effectiveness of the techniques

Table 3
Mapping from ids to actual methods and constraints.

Identifier	Definition
m1	run()
m2	handleServerMessage()
m3	handleServiceMessage()
m4	getName()
m5	getApplication()
m6	ServiceManager.getInstance().getService()
m7	getUser()
m9	Store.getServiceStore().getService()
m10	isAuthorizedService()
m11	sendAck()
m12	process()
c1	!isAlreadyConnected
c2	!isAuthenticationMessage
c3	!isServerInfoMessage
c4	message.getApplication().equals("Server")
c5	userName
c6	serviceName
c7	s
c8	user
c9	service
c10	isAuthorizedService

according to the thoroughness of the test suites available for executing the program under analysis. We do not intend to study the sensitivity of the techniques to the choice of the parameter *k*, especially because there exist a number of studies that already shows that values of *k* between 2 or 3 are good choices when the inferred models represent sequences of method invocations produced by an application (Reiss and Renieris, 2001; Cook and Wolf, 1998; Mariani et al., 2011b). In line with these results, we run the inference techniques with a value of *k* equals to 2.

To study the effectiveness of the techniques according to different levels of completeness of the traces, we inferred the models from traces that satisfy three coverage criteria: state coverage, transition coverage, and 2-transition coverage. The coverage criteria are expressed with respect to the reference model. Intuitively, *state coverage* corresponds to a sparse set of executions that covers all the states in the model but do not exercise all the method invocations, (i.e., transitions). *Transition coverage* corresponds to a good set of executions that samples all methods invocations, but does not invoke the methods in all the possible execution contexts, for instance loops are not necessarily executed multiple times. 2-*transition coverage* corresponds to a thorough test suite that samples each method invocation at least twice, increasing the chance to execute method invocations in different contexts. We do not consider stronger coverage criteria because they would represent unrealistic scenarios with fairly complete sets of executions that are extremely hard to obtain in the practice.

We produce traces from a reference model by randomly traversing the EFSA from the initial state to a final state. If a final state has outgoing transitions, we randomly choose to either end the trace or continue producing a longer trace. When traversing a transition that has one or more parameters, we randomly generate a value that satisfies the constraints associated with the transition. We continue generating traces until the selected coverage criterion is satisfied. We produce traces based on a random strategy to avoid obtaining empirical data biased by the strategy used to cover models.

A final remark is about the constraints considered in the reference EFSAs. We do not consider algebraic constraints that include operators that cannot be inferred by gkTail. We are not interested in investigating the complexity of the constraints that gkTail can infer (which is evaluating the constraint generation engine, that is Daikon (Ernst et al., 2001)), but rather we intend to evaluate the effectiveness of gkTail to both associate the right constraints at the right transitions and merge the right states when the evaluation of their futures include the comparison of the constraints, which only depend on the heuristics implemented in gkTail. Thus, the reference EFSAs only include constraints that can potentially be inferred by Daikon, and hence gkTail.

In the following, we describe the empirical process that we follow to answer the three research questions investigated in this paper.

4.1. Research question 1

Do inference techniques producing extended FSAs generate models that can better identify legal behaviors as compared to those producing simple FSAs?

To answer this research question, we compute the recall (also known as true positive rate) of the models inferred with kTail, kBehavior, gkTail and KLFA, for all the coverage levels and reference models considered in our empirical assessment. Recall is a common measure used in information retrieval (Manning et al., 2008). In our setting it measures the ability of the model in identifying correct behaviors. The intention here is to verify whether considering models more complex than simple FSAs results in an increment or a loss in recall.

To compute recall, we followed the process shown in Fig. 5. We first generated three training sets from the reference models. Each of the training sets satisfies one of the three coverage criteria described earlier. We applied the four model-generation techniques considered in this study to infer models from the training sets. To compute the recall of the inferred models we generated a new set of traces, namely the evaluation traces, from the reference model. To deeply compare the reference and inferred models, the evaluation traces satisfy 10-transition coverage, i.e., traces cover each transition at least 10 times. We finally computed the recall of the inferred models as the fraction of the evaluation traces that are correctly accepted by the inferred models. The more traces are accepted, the higher the recall of the inferred model is.

4.2. Research question 2

Do inference techniques producing extended FSAs generate models that can better reject illegal behaviors as compared to those producing simple FSAs?

In order to answer this research question, we measure how many illegal behaviors are erroneously included into the models inferred by kTail, kBehavior, gkTail and KLFA. In our context, a behavior can be illegal for any of the two following reasons: it includes an illegal sequence of operations, or it includes an illegal parameter value. Since only extended models can detect illegal parameter values, we evaluate the quality of the models with respect to these two classes of illegal behaviors separately. In this way, we can specifically compare and measure how much the capability of rejecting illegal behaviors due to the presence of illegal parameter values, which is a unique capability of EFSAs, impacts on the capability to reject illegal behaviors due to the presence of illegal sequences of operations.

Fig. 6 shows the process that we followed to investigate this research question. To measure how many illegal sequences of operations are erroneously included into inferred models we compute precision (Manning et al., 2008). The precision quantifies the percentage of illegal behaviors that have been (erroneously) incorporated into the inferred models. A low precision indicates that many illegal behaviors are present in the inferred model, thus compromising its rejection capability. A high precision indicates that few illegal behaviors are present in the model, thus the model has an excellent rejection capability. Precision is computed by generating traces from the inferred models and checking the fraction of those traces that are accepted by the reference model. To obtain accurate precision value, we generate traces until all transitions in the inferred model are covered 10 times. Thus, the generated traces cover each single operation in multiple usage contexts. When generating traces from extended models, we only generate traces with event sequences ignoring the parameter values, which are studied separately.

Furthermore, to study the capability to reject illegal behaviors due to illegal parameter values we measure specificity. Specificity (also known as true negative rate) (Manning et al., 2008) is obtained by producing illegal traces from the reference model and checking the fraction of traces that are correctly rejected by the inferred models. To produce the traces with illegal parameter values, we mutated the reference automata by producing automata that exactly match the original ones with the exception of a randomly selected constraint that is negated. We then generate traces from the automata with the requirement to cover each transition 10 times. Finally, we filter those traces that do not traverse the mutated constraint, and we measure the fraction of traces correctly rejected by the inferred automata. We perform this evaluation only for gkTail and KLFA because we know in advance that kTail and kBehavior cannot reject any trace based on only the parameter values.

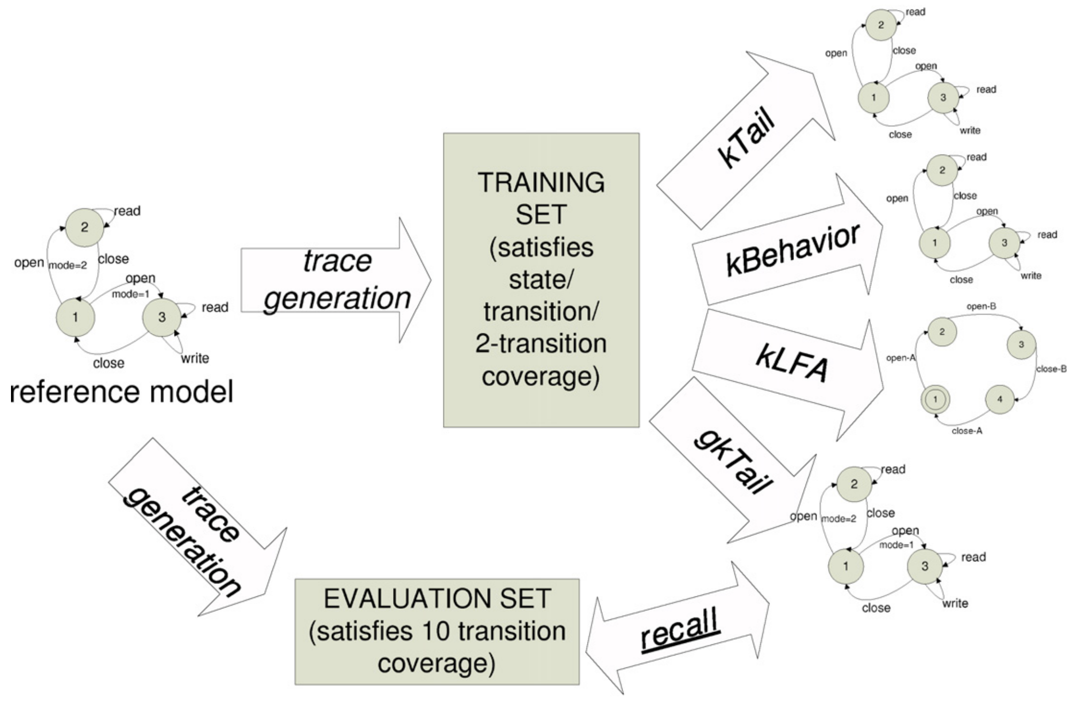


Fig. 5. The empirical process for the computation of recall.

4.3. *Research question 3*

What is the performance difference between the generation and the checking of extended FSAs as compared to those of simple FSAs?

To answer this research question, we measured both the time spent by the techniques to perform the inference and the time required to check traces. These measures give hints about the possible usages of the techniques. For instance, techniques that require a long time to generate models can be used when the set of the traces is quite stable over time, and cannot be effectively used when this set frequently changes. Similarly, techniques that require a substantially long time to check traces can be used for offline analysis, but might not be suitable to be used in a deployed system since they can compromise performance. Note that the reported inference time includes all the operations necessary to obtain the models from the traces. For example, in the case of KLFA, this includes the time needed to run the rewriting strategy.

5. Toolset

In order to support the empirical study described in this paper we developed three new tools (the *trace generator*, the *mutant generator* and the *targeted trace generator*) and extended the QUARK platform (Lo and Khoo, 2006). Fig. 7 shows the tools that we used in the empirical validation.

The trace generator tool can automatically generate traces from EFSAs according to a given coverage criterion. Trace generation is achieved by randomly walking the EFSA. When a value that satisfies a constraint needs to be generated, the trace generator randomly

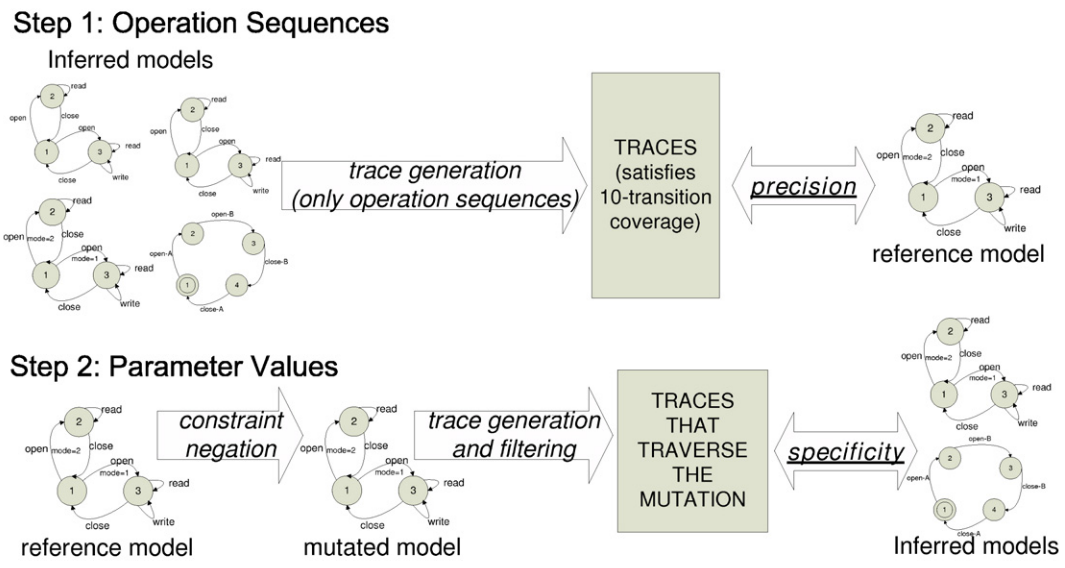

Step 1: Operation Sequences
Inferred models
trace generation (only operation sequences)
TRACES (satisfies 10-transition coverage)
precision
reference model
Step 2: Parameter Values
reference model
constraint negation
mutated model
trace generation and filtering
TRACES THAT TRAVERSE THE MUTATION
specificity
Inferred models

Fig. 6. The empirical process for the computation of precision and specificity.

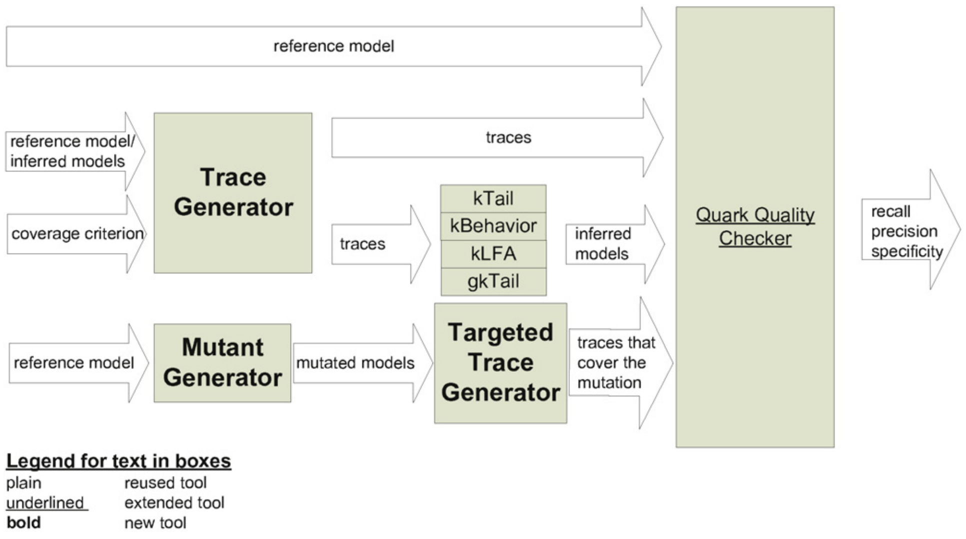


Fig. 7. The toolset used in the empirical evaluation.

produces a value that satisfies it. The trace generator is used in our empirical validation in three different cases: (1) to generate the traces from the reference models to build the inferred models with inference engines, (2) to generate traces from the reference EFSA to compute recall, and (3) to generate traces from the inferred models to compute precision.

The mutant generator is a tool that can produce a mutated EFSA by randomly negating a constraint in the EFSA. The tool is executed in our empirical validation to build the mutated EFSAs used to compute specificity.

The targeted trace generator is equivalent to the trace generator with the exception that it generates only traces that cover the mutated constraint. In our empirical validation this tool is used to generate traces from mutated EFSAs to compute specificity.

QUARK is a tool that can compute various goodness measures given a set of traces and a FSA (Lo and Khoo, 2006). We extended the QUARK framework to support the computation of precision, recall and sensitivity for EFSAs, and we used the extended version of QUARK to compute these metrics in our empirical validation.

Finally, the inference algorithms compared in this work have been integrated in our toolset as black box components.

Interested readers can find links to the tools used for the comparison and the models used in our case study at the following web page: http://www.lta.disco.unimib.it/papers/model_assessment/.

6. Empirical results

In this section we present and discuss the empirical result obtained according to the processes described in Section 4.

6.1. Research question 1

To answer the first research question we computed the recall of the inferred models for a total of 30 cases per technique, obtained by experimenting with the three coverage levels (state, transition, and 2-transition) for each of the 10 case studies. Table 4 specifies the minimum, maximum and average number of traces generated in the experiments according to the coverage level. Note that 10-transition coverage is not used for inferring the models, but it is used to generate the evaluation set.

Table 5 shows the empirical results about recall. Fig. 8 shows the box-plot of the recall of each technique, grouped by coverage criteria. A box-plot graphically represents a distribution of values

Table 4
Min, max and average number of generated traces.

	State coverage	Transition coverage	2-Transition coverage	10-Transition coverage
Minimum	11	15	23	41
Maximum	215	393	1405	1929
Average	64.6	120.6	279.8	700.6

with a rectangle, which is delimited by the first and third quartile, a solid line, which indicates the median, and two whiskers, which end with the minimum and maximum values. Gray boxes show the recall of techniques that ignore data-flow information (i.e., kTail and kBehavior), while white boxes show the recall of techniques that infer data-flow information (i.e., gkTail and KLFA).

We can notice from the plot that considering data-flow information causes a loss of recall. In the case of gkTail, compared to kTail, the loss is moderate (0.03 on average), while it is substantial comparing KLFA to kBehavior (0.16 on average). We used the Mann–Whitney U test (Wohlin et al., 2000) (which does not expect the empirical data to follow normal distribution) to check if the difference in recall between the techniques that ignore data-flow

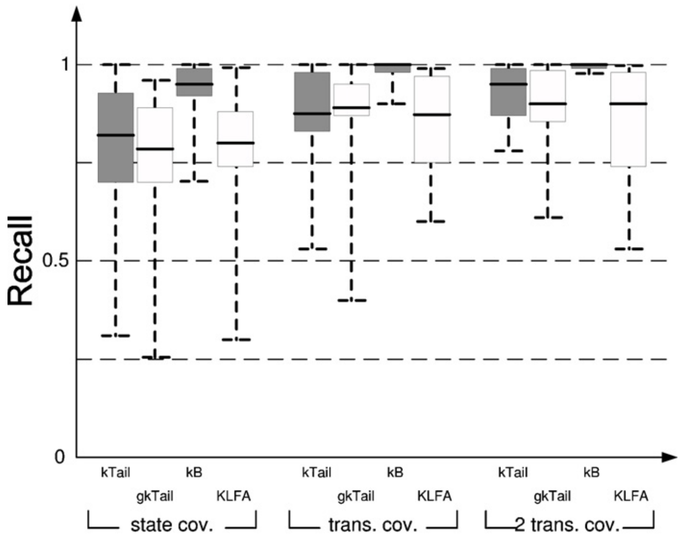


Fig. 8. Recall for different coverage levels.

Table 5
Empirical results about recall.

Application	kTail			gkTail		
	State	Transition	2-Transition	State	Transition	2-Transition
JFreeChart	0.31	0.54	0.82	0.26	0.4	0.61
Lucane	0.79	0.87	0.92	0.79	0.87	0.91
ThingamaBlog	0.92	0.98	0.99	0.91	0.97	0.99
Jeti	0.8	0.83	0.95	0.76	0.87	0.9
Columba	0.68	0.88	0.85	0.68	0.88	0.85
Open Hospital	1	1	1	0.95	1	1
Rapid Miner	0.76	0.85	0.93	0.74	0.85	0.89
Heritrix	1	1	1	0.98	0.99	1
JArgs	0.94	0.98	0.98	0.79	0.9	0.89
Tagsoup	0.6	0.65	0.77	0.6	0.67	0.73
Median	0.82	0.87	0.95	0.79	0.89	0.9

Application	kBehavior			KLFA		
	State	Transition	2-Transition	State	Transition	2-Transition
JFreeChart	0.71	1	1	0.3	0.59	0.7
Lucane	0.92	0.98	0.97	0.81	0.83	0.88
ThingamaBlog	0.94	1	1	0.88	0.99	0.99
Jeti	0.95	0.9	0.98	0.8	0.68	0.82
Columba	1	1	1	0.46	0.76	0.54
Open Hospital	1	1	1	0.9	0.92	0.97
Rapid Miner	0.92	0.98	0.99	0.84	0.97	0.91
Heritrix	1	1	1	0.99	0.99	0.99
JArgs	0.99	1	1	0.74	0.75	0.71
Tagsoup	0.89	1	1	0.74	0.97	0.99
Median	0.95	1	1	0.81	0.87	0.9

information and the techniques that consider data-flow information is significant. Since we do not know in advance the effect of data-flow information we used a two-tailed test. We considered *p*-values below 0.05 to be significant. Table 6 summarizes the results. The addition of algebraic constraints to kTail introduces a statistically significant loss of recall only when traces satisfy 2-transition coverage. This result suggests that kTail can better take advantage of the availability of many traces than gkTail, while this capability has no statistically significant effect when traces satisfy state or transition coverage. Inferring universally quantified constraints has a more negative effect on the inference process. In fact KLFA consistently produced FSAs with worse recall than kBehavior, and this result is statistically significant for all coverage levels.

The size of the boxes, which represents how varying the recall is, increases when introducing universally quantified constraints in kBehavior. In fact the recall of KLFA varies a lot compared with the recall of kBehavior, which is quite close to its median value. The same effect is not observable for gkTail. The higher variability of KLFA indicates that it is harder to correctly learn universally quantified constraints, as performed by KLFA, than learning algebraic constraints on data values, as performed by gkTail. This is particularly true when the data used for the inference comes from method executions, where thousands of different methods are executed each with a different set of parameters.

If we look at the recall while the level of coverage increases, we have another interesting result. The recall and the stability of the results improve faster for techniques that do not consider data-flow information (boxes are smaller and closer to the top of the diagram). The slow improvement of gkTail and KLFA is due to the amount of extra information needed by those techniques to produce accurate models: covering each transition twice is not always sufficient to discover enough data-flow information to significantly improve the model. Intuitively, the same method invocations should be executed several times with different parameter values to produce enough data to let the inference techniques understand that these values satisfy some constraints or reoccur according to a pattern.

Our empirical study about recall also provides information about the absolute effectiveness of the techniques when varying the number of traces (i.e., test cases). When traces are generated with state coverage criterion, kBehavior is the only technique with its box entirely above 0.75. The results are more encouraging for all the techniques when traces are generated with transition coverage criterion (the boxes of three out of the four techniques are entirely above 0.75), and are good for 2-transition coverage. These results suggest that FSAs with good recall can be effectively extracted even from a sparse set of executions, while EFSAs require thoroughly tested software to derive models with good recall, that is, it requires at least a set of execution traces that covers the program under test following the transition coverage criterion. An important research direction is thus the development of techniques that can produce test cases that well exercise the program under analysis, as investigated in Dallmeier et al. (2010).

We have another interesting result from the detailed data reported in Table 5. We expect inference techniques produce better models when more traces are available. In fact, if the additional data are not suitably generalized, some behavior might be improperly represented in the model, especially if data-flow information must be handled in addition to event sequences. This is the case for KLFA when analyzing Jeti, Columba, Rapid Miner and JArgs, where an increased coverage sporadically causes the generation of models with poorer recall. These sporadic losses of recall are caused by the identification of imprecise recurrent patterns. When KLFA infers a model from a limited number of traces (e.g., state or transition

Table 6
Test of significance for difference in recall (*p*-value <0.05 is considered significant).

	kTail/gkTail		kBehavior/KLFA	
	p-Value	Sig	*p*-Value	Sig
State cov	0.07	No	<0.01	**Yes**
Trans cov	0.39	No	<0.01	**Yes**
2 Trans cov	0.03	**Yes**	<0.01	**Yes**

Table 7
Average recall values according to model complexity.

	kTail	gkTail	kBehavior	KLFA
Easy (CC < 10)	0.84	0.83	0.97	0.73
Medium (10 ≤ CC ≤ 15)	0.96	0.94	0.99	0.9
Hard (CC > 15)	0.62	0.55	0.93	0.72

coverage), it identifies few clear patterns. While when KLFA infers the model from a higher number of traces (e.g., transition or 2-transition coverage) the amount of available data increases causing the incidental identification of patterns that do not hold in general. Having even more data would have eliminated these cases. Since the likelihood to have a pattern represented in the model is dependent on the amount of data that support those patterns, it is natural to observe some statistical fluctuations in KLFA results.

Finally, we investigated the recall of the four techniques on various model complexity levels. Table 7 shows the average value of recall with respect to the complexity of the model (we use the cyclomatic complexity as indicator of complexity). We can notice that inference techniques based on state merging heuristics (i.e., kTail and gkTail) handle easy and medium complexity cases well, but are weak on the hard cases. On the contrary, inference techniques based on behavioral patterns (i.e., kBehavior and KLFA) degrade gracefully with model complexity. Finally, all techniques consistently inferred models with better recall for medium complexity cases, indicating that inference techniques can inappropriately generalize simple and complex models.

6.2. Research question 2

To answer the second research question, we measured both the fraction of legal sequences of operations that can be generated from the inferred models and the fraction of operation sequences that include illegal parameter values that are correctly rejected by the inferred models. The former is known as precision and it is computed for all the techniques. The latter is known as specificity and it is computed for techniques that can infer EFSAs.

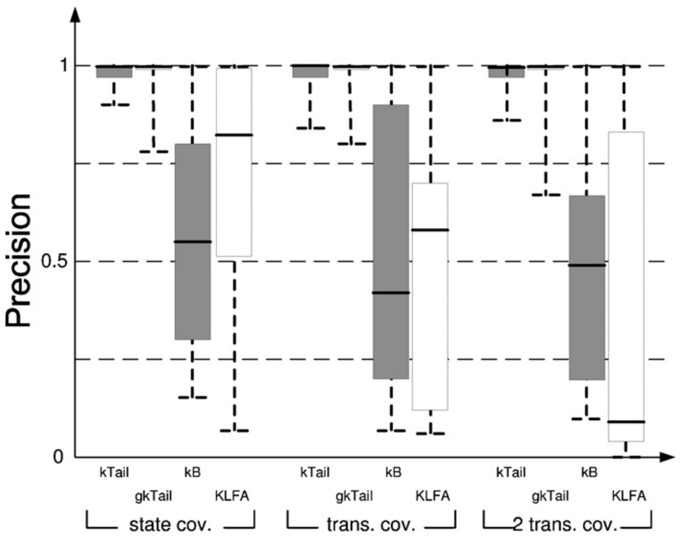


Fig. 9. Precision for different coverage levels.

Table 8 shows the empirical data about precision. Fig. 9 shows the box plot of precision for all techniques, grouped by coverage criteria.

The empirical data about precision suggests that extending techniques for the inference of FSAs with the capability of handling data-flow information tends to improve precision, with the exception of KLFA when applied to traces collected with 2-transitions coverage. To confirm this intuition we employ the Mann–Whitney *U* test to check if the differences in precision between kTail and gkTail, and between kBehavior and KLFA are statistically significant. Table 9 shows the results.

Table 8
Precision (limited to operation sequences).

Application	kTail			gkTail		
	State	Transition	2-Transition	State	Transition	2-Transition
JFreeChart	1	1	1	1	1	0.67
Lucane	1	1	0.99	1	1	1
ThingamaBlog	1	1	1	1	1	1
Jeti	1	1	1	1	1	1
Columba	1	1	1	1	1	1
Open Hospital	1	1	1	1	1	1
Rapid Miner	0.92	0.94	0.86	1	1	1
Heritrix	0.97	0.98	0.97	0.78	0.8	0.8
JArgs	0.91	0.82	0.87	1	1	0.99
Tagsoup	0.98	0.99	0.98	1	0.99	0.99
Median	1	1	0.99	1	1	1

Application	kBehavior			KLFA		
	State	Transition	2-Transition	State	Transition	2-Transition
JFreeChart	0.41	0.23	0.49	1	0.06	0.09
Lucane	1	0.97	0.47	1	1	1
ThingamaBlog	0.29	0.19	0.96	0.96	0.1	0.89
Jeti	0.86	1	1	0.96	1	1
Columba	0.83	1	0.58	1	0.52	0.66
Open Hospital	0.69	0.07	0.1	0.54	0.68	0
Rapid Miner	0.38	0.18	0.1	0.07	0.19	0
Heritrix	0.15	0.23	0.44	0.17	0.08	0.08
JArgs	0.7	0.62	0.12	0.7	0.65	0.11
Tagsoup	0.29	0.7	0.68	0.5	0.71	0.04
Median	0.55	0.42	0.48	0.83	0.59	0.1

Table 9
Test of significance for difference in precision.

	kTail/gkTail		kBehavior/KLFA	
	p-Value	Sig	*p*-Value	Sig
State cov	0.02	**Yes**	0.01	**Yes**
Trans cov	0.07	No	0.33	No
2 Trans cov	0.22	No	0.02	**Yes**

We can notice that the improvement on precision produced by gkTail and KLFA over kTail and kBehavior respectively are statistically significant when traces satisfy state coverage. Thus, when few traces are available the incorporation of data-flow information in the models is beneficial for precision. The difference in precision is not anymore significant when the number of the traces increases. There is one exception, that is KLFA. In fact, KLFA shows a loss of precision compared to kBehavior that is statistically significant when traces satisfy 2-transitions coverage.

If we look at the results in absolute terms, it is evident that gkTail and kTail have better precision than the other techniques. This suggests that when precision is important compared to other aspects (e.g., recall), one of these two algorithms should be preferred, depending on the necessity to have data-flow information incorporated in the inferred model.

We can also notice that increasing the number of available samples does not evidently improve the precision of the techniques. The detailed data reported in Table 8 also confirms this trend: the precision is not strictly increasing when the number of available traces increases. In a few cases the precision decreases even if more traces are available. This suggests that merely increasing coverage is not enough to guarantee better results, at least in term of precision. These fluctuations on the results are due to the generalization capabilities of the techniques. This over-generalization can be avoided by further increasing the number of traces, that is not easy to do in practice, and by extracting negative traces, for instance by using static analysis to identify infeasible behaviors (Mariani et al., 2010). Investigating techniques to extract negative traces from systems is a promising research direction because a number of learners can provide extremely good effectiveness when both positive and negative traces are produced. In general, it is necessary to further investigate the relationships between the kind of coverage that is achieved and the quality of the inferred model.

Finally, we investigated the precision of the four techniques according to the complexity of the behavior that is analyzed. Table 10 shows the average value of the precision with respect to the complexity of the model. We can notice that inference techniques based on state merging heuristics (i.e., kTail and gkTail) handle well models of any complexity, while the precision of inference techniques based on behavioral patterns (i.e., kBehavior and KLFA) significantly degrades with increasing model complexity.

Table 10
Average precision values according to model complexity.

	kTail	gkTail	kBehavior	KLFA
Easy ($CC < 10$)	0.99	1	0.86	0.9
Medium ($10 \leq CC \leq 15$)	0.95	0.96	0.35	0.35
Hard ($CC > 15$)	0.99	0.94	0.46	0.4

Table 11 shows the empirical data about the specificity of the inferred models, computed with traces that should be rejected because they include illegal parameter values. We only report data about gkTail and KLFA because the specificity is 0 for kTail and kBehavior. We can notice that models inferred by gkTail have high-specificity in all the cases, while models inferred by KLFA have widely varying specificity, from extremely low (0) to extremely high (1).

The specificity of models obtained with gkTail indicates that most of the algebraic constraints are correct and annotate the right transition. On the contrary, KLFA well represented only a subset of the constraints, with a decrease on the specificity when traces satisfy transition coverage rather than state coverage. This difference on the results are affected by two main factors: the kind of models used for the inference and the way KLFA works.

A major aspect affecting the value of the specificity is the kind of data-flow information represented in the models. While gkTail can potentially identify all the constraints in the reference models, this is not true for KLFA. In fact, KLFA represents the way parameter values reoccur across method invocations and does not represent the concrete values that are allowed for a parameter in a given method invocation. Thus KLFA can successfully identify that a value is illegal only if such a value occurs in a data-flow pattern, which is not always the case. In addition, KLFA can lose specificity when the number of traces increases because KLFA automatically discards the recurrence patterns that do not apply well to a large subset of the traces. In other words, KLFA favors the identification of few relevant patterns to the identification of many, but noisy, patterns. Thus the lower level of specificity is somehow expected and inherent to an the algorithm that produces only relevant patterns. This strategy is demonstrated to be useful when KLFA is used for the analysis of log files (Mariani and Pastore, 2008), but apparently is less effective when applied to traces consisting of method invocations.

In conclusion, gkTail has a higher specificity than KLFA when applied to our reference models. However, both techniques are important because gkTail and KLFA focus on complimentary aspects: algebraic constraints versus universally quantified constraints, and they can detect different kinds of anomalous behaviors.

Table 11
Specificity (limited to parameter values).

	gkTail			KLFA		
	State	Transition	2-Transition	State	Transition	2-Transition
JFreeChart	0.79	1	1	0.93	0.68	0.51
Lucane	1	1	1	0.67	0.64	0.64
ThingamaBlog	0.94	1	1	0.6	0.07	0.01
Jeti	1	1	1	0.55	0.66	0.47
Columba	1	1	1	0.63	0.24	0.55
Open Hospital	1	0.91	0.91	1	1	1
Rapid Miner	1	1	1	0.96	0.42	0.46
Heritrix	0.96	1	1	0.01	0.02	0.02
JArgs	0.94	0.99	0.99	1	1	1
Tagsoup	0.93	0.98	1	0.96	0.07	0
Med spec	1	1	1	0.67	0.64	0.52

Table 12
Average time to infer and check models.

	kTail	kBehavior	gkTail	KLFA
Inference	5 s	2 s	38 min	3 s
Checking	9 s	8 s	13 s	4 s

6.3. Research question 3

To answer the third research question we computed the average time spent by kTail, kBehavior, gkTail and KLFA to produce the models and evaluate traces for acceptance (see Table 12). The average is computed over the 30 cases (10 case studies analyzed with 3 coverage levels each) considered for each technique. For the evaluation, we used a standard desktop computer (Intel Core Duo 2 GHz, 2 GB RAM).

The inference of the models is fast for all the techniques (on average less than 5 s to process hundreds of traces), with the exception of gkTail that required an average of 38 min to generate the models. The long inference time is due to the many executions of the Daikon learner (Ernst et al., 2001) that is integrated in gkTail and is used to produce the algebraic constraints included in the PTA. Thus gkTail can be effectively used only if the traces used to generate the models are quite stable over time, while the other techniques can be used more flexibly.

All the techniques can check traces fast. We have an exceptional time for KLFA, while we have longer times for kBehavior and kTail. This difference is mostly related to KLFA that integrates an optimized version of kBehavior, while the implementation of kTail and the stand-alone version of kBehavior are not optimized. We have reason to believe that these differences would not occur if all the algorithms had been implemented in an optimized way. gkTail requires more time than the other techniques to check traces because it must execute a constraint solver to verify if data values satisfy constraints. We can also observe that checking constraints impacts the performance, but it has no dramatic effect, according to our empirical experience.

7. Final remarks

The main conclusions that can be derived from our empirical work are:

Adding algebraic constraints to FSAs does not compromise quality but negatively affects performance: gkTail does not introduce significant differences on the quality of the inferred models compared to kTail: the only statistically significant differences have been observed for recall when many traces are available (i.e., small loss of recall when traces satisfy 2-transition coverage) and for precision when few traces are available (i.e., small improvement of precision when traces satisfy state coverage). gkTail produces models with high specificity, thus algebraic constraints are well captured by the inference engine. Unfortunately, gkTail has an extremely long inference time compared to other techniques: on average, half an hour versus a few seconds.

Adding universally quantified constraints to FSAs negatively affects recall: KLFA introduces a statistically significant loss of recall at all coverage levels compared to kBehavior. Precision is significantly better for traces that satisfy state coverage, but it is significantly worse for traces that satisfy 2-transitions coverage. KLFA does not significantly affect the inference time. These results indicate that recurrence patterns can be added to FSAs without compromising the inference time, but negatively affecting the quality of the model.

Adding traces is important but not enough: in the empirical validation, increasing the number of traces positively correlates to recall. We notice that the improvement in recall is higher for the inference of simple FSAs compared to that of extended FSAs when the number of traces is increased. On the contrary, there is no clear trend in precision. This is due to the difficulty in controlling over-generalization when only positive samples are available. The over-generalization should be controlled with negative samples, which are typically hard to find in practice. The study of techniques that extract negative traces is an important research direction for the future.

State merging heuristics have good precision but poor recall: In the empirical validation, techniques based on a state-merging heuristic (i.e., kTail and gkTail) produced models with good precision independently from the complexity of the model, while recall dropped for the hard cases.

Behavior merging heuristics have good recall but poor precision: In the empirical validation, techniques based on a behavior-merging heuristic (i.e., kBehavior and KLFA) produced models with good recall independent of the complexity of the models, while precision dropped for the medium and hard cases.

State coverage is enough to produce useful models: the models inferred from traces that satisfy state coverage have good recall and precision, and results generally improve with transition coverage. Such empirical results suggest that the inference techniques investigated in this study can be applied in the practical cases where coverage is often partial.

"Hard" models are hard: inferring hard models, that is models with CC > 15, from positive traces is challenging. In fact, state- and behavior-merging heuristics have low recall and precision respectively, when inferring hard models.

8. Threats to validity

A threat to validity is the generalization of our results. We have tried to address this threat and improve the generalizability of our results in several ways. First, we analyzed 10 models that are both realistic (they have been extracted from real software systems) and relevant to our comparative assessment study (they include an interplay of aspects relevant to all the compared techniques). The number of considered models is larger than most of prior studies (Ramanathan et al., 2007; Walkinshaw and Bogdanov, 2008; Lo and Maoz, 2009; Thummalapenta and Xie, 2009; Zhong et al., 2009). Moreover, we selected models from multiple software systems rather than extracting multiple models from few systems to avoid the generation of results biased by a specific programming style. The set of samples used for the inference have been produced taking into account the partial coverage that test suites usually provide in practice. We thus believe that the empirical experience reported in this paper provides relevant insights on the inference of automata from software and highlights important aspects that should be taken into account when using model inference to support testing and analysis techniques. However, a different set of models might produce different results.

Another threat to the validity of the empirical validation is the use of a single value for the parameter *k*, which influences the behavior of all the compared techniques. Studying the impact of the choice of *k* to the results even if interesting, is outside the scope of this work. Past studies have shown that small values of k, such as 2 or 3, are good choices. In this study we set *k* to 2.

An additional threat is the exclusion of the algebraic constraints that cannot be addressed with Daikon from the reference EFSAs used for the empirical study. Still, we believe Daikon's constraints characterize most (or at least a significant proportion) of useful constraints exhibited in various software systems. A final threat is related to the correctness of the tools that we used for the empirical validation. We manually validates the tools with a number of small artificial models that replicate several common situations.

Moreover, we manually checked the correctness of the results for a random selection of the traces generated in the empirical study. These activities gave us confidence on the correctness of the results.

9. Related work

In this section we discuss closely related studies on learning FSAs, learning other models, and comparative and empirical studies.

9.1. Learning FSAs

Techniques that generate models from execution traces have been used to support a number of software engineering tasks. FSAs is one of the most commonly inferred models (Biermann and Feldman, 1972; Mariani and Pezzè, 2007; Dallmeier et al., 2006; Ammons et al., 2002; Raffelt and Steffen, 2006; Walkinshaw et al., 2007; Antunes et al., 2011; Bertolino et al., 2009; Schneider et al., 2010). FSAs are simple and useful abstractions that can well represent the possible ordering of events generated by programs. However, FSAs completely neglect aspects related to data and the interplay between data and event sequences.

Some novel techniques define mechanisms to infer FSAs extended with information about the data-flow (Lorenzoli et al., 2008; Mariani and Pastore, 2008; Krka et al., 2010). On a conceptual point of view extended FSAs can indeed represent all the behaviors that can be represented with simple FSAs, with the addition of behaviors that cannot be represented. However, the generation of models more complex than simple FSAs is challenging and higher expressiveness can introduce inaccuracies.

In this paper we empirically compared the quality of the extended FSAs produced by gkTail (Lorenzoli et al., 2008) and KLFA (Mariani and Pastore, 2008) with the simple FSAs produced by kTail (Biermann and Feldman, 1972) and kBehavior (Mariani and Pezzè, 2007). Other algorithms are related with the algorithms empirically compared in this paper.

The work by Krka et al. (2010) could mine FSAs extended with algebraic constraints. However, the approach has only been evaluated on one relatively small example and its scalability to larger cases is still unclear. Furthermore, although extended FSAs could be derived from Krka et al's learned FSAs, the main goal of Krka et al.'s work is not to extend FSAs with algebraic constraints rather to use algebraic constraints to improve the quality of the learned simple FSAs.

There are many simple FSA learners (Biermann and Feldman, 1972; Mariani and Pezzè, 2007; Dallmeier et al., 2006; Ammons et al., 2002; Raffelt and Steffen, 2006; Walkinshaw et al., 2007; Antunes et al., 2011; Bertolino et al., 2009; Schneider et al., 2010). The approach ranges from passive (i.e., only analyze execution traces) to active (i.e., ask queries to users or generate test cases), such as (Raffelt and Steffen, 2006; Walkinshaw et al., 2007; Bertolino et al., 2009). These learners also analyze different kinds of execution traces, from standard program execution traces, to network traces (Antunes et al., 2011), WSDL descriptions (Bertolino et al., 2009), and system logs (Schneider et al., 2010). We do not analyze many of these studies because there are no corresponding extensions of these approaches that can mine extended FSAs, either by the incorporation of algebraic or universally quantified constraints. The incorporation of these constraints in the mining process is not trivial. Thus in this study we focus on k-Tail (Biermann and Feldman, 1972) and k-Behaviors (Mariani and Pezzè, 2007) which have been extended to mine FSAs extended with algebraic and universally quantified constraints.

9.2. Learning other models

Several techniques can infer models differing from FSAs to describe relations between events. Temporal relations between events can be represented by other models, such as patterns of events (Lo et al., 2007; Safyallah and Sartipi, 2006) and temporal logic rules (Yang et al., 2006; Lo et al., 2008). In this work, we focus on techniques explicitly deriving FSAs, with a particular emphasis to study the costs and tradeoffs of incorporating data-flow information within FSAs. Comparing the effectiveness of different kinds of models in capturing the relations between events is an interesting research direction that we plan to investigate in the future.

9.3. Comparative and empirical studies

The closest empirical study available, in addition to the empirical study reported in this paper, shows that software programs include behaviors that could only be represented by extended models (Lorenzoli et al., 2008; Mariani and Pastore, 2008). However, neither empirical results about the overall quality of the extended models nor comparative empirical results of corresponding techniques that infer simple FSAs and extended FSAs have been reported yet. The results obtained in this paper provide an early picture of the tradeoffs between the different learners and give testers quantitative information to guide them into the choice of the inference technique.

There are also other studies that compare the quality of learning techniques that infer simple FSAs. Lo et al. propose a technique named QUARK that measure the quality of a learned automaton based on its language similarity with the target automaton (Lo and Khoo, 2006). Bogdanov and Walkinshaw (2009) and Pradel et al. Pradel et al. (2010) propose techniques that measure the quality of a learned automaton based on its structural similarity with the target automaton. In this study, we extend the above studies by comparing the quality of learners that infer simple FSAs to those that infer extended FSAs (i.e., FSAs extended with algebraic and universally quantified constraints).

There are also interesting surveys on program analysis techniques; one of those is a survey by Cornelissen et al. (2009) which focuses on the usage of dynamic analysis to aid program comprehension. In this work, we perform a comparative study on a subtopic of dynamic analysis namely behavioral model inference.

10. Conclusions

A number of techniques can generate behavioral models from execution traces. A kind of model that is commonly used to represent the behavior of a software program and that can be easily inferred from execution traces is FSA. In the recent years, it has been shown that several program behaviors that cannot be represented with FSAs, can be inferred and represented with extended FSA (Lorenzoli et al., 2008; Mariani and Pastore, 2008). Thus, extended FSAs are more expressive than simple FSAs. However, learning extended FSAs is challenging and the inferred models can be inaccurate. In this paper, we presented an early empirical comparative study that investigates both the tradeoffs between techniques that infer simple FSAs and those that infer extended FSAs and the effectiveness of these techniques when varying the number of available traces. In particular, we evaluated kTail (Biermann and Feldman, 1972), kBehavior (Mariani and Pezzè, 2007), gkTail (Lorenzoli et al., 2008) and KLFA (Mariani and Pastore, 2008) with a set of 10 case studies extracted from real software systems.

The empirical results show the tradeoffs between techniques. The interpretation of the collected data highlights costs, drawbacks and benefits of each model inference solution, with specific references to the types of traces used for the inference. The resulting discussion provides indications that can help software engineers in the choice of proper model inference solutions.

Acknowledgment

This work is partially supported by the European Community under the 7th FP for project PINCETTE.

References

Ammons, G., Bodík, R., Larus, J.R.,2002. Mining specifications. In: Proceedings of the Symposium on Principles of Programming Languages. ACM.

Antunes, J., Neves, N.F., Veríssimo, P.,2011. Reverse engineering of protocols from network traces. In: Proceedings of the Working Conference on Reverse Engineering. IEEE.

Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.,2009. Automatic synthesis of behavior protocols for composable web-services. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM.

Biermann, A., Feldman, J., 1972. On the synthesis of finite-state machines from samples of their behaviour. IEEE Transactions on Computers 21, 591–597.

Bogdanov, K., Walkinshaw, N.,2009. Computing the structural difference between state-based models. In: Proceedings of the Working Conference on Reverse Engineering. IEEE.

Columba. sourceforge.net/projects/columba (accessed 2011).

Cook, J., Wolf, A., 1998. Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology 7, 215–249.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering 35, 684–702.

Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.,2006. Mining object behavior with ADABU. In: Proceedings of the International Workshop on Dynamic Systems Analysis. ACM.

Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.,2010. Generating test cases for specification mining. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM.

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27, 99–123.

Hangal, S., Lam, M.S.,2002. Tracking down software bugs using automatic anomaly detection. In: Proceedings of the International Conference on Software Engineering. ACM.

Henkel, J., Diwan, A.,2003. Discovering algebraic specifications from java classes. In: Proceedings of the European Conference on Object Oriented Programming. Springer.

Heritrix. http://crawler.archive.org/ (accessed 2011).

Jargs command line option parsing suite for Java. http://jargs.sourceforge.net/ (accessed 2011).

Jeti. jeti.sourceforge.net (accessed 2011).

Jfreechart. www.jfree.org/jfreechart (accessed 2011).

Krka, I., Brun, Y., Popescu, D., Garcia, J., Medvidovic, N.,2010. Using dynamic execution traces and program invariants to enhance behavioral model inference. In: Proceedings of the International Conference on Software Engineering (NIER Track). ACM.

Lo, D., Khoo, S.-C.,2006. QUARK: empirical assessment of automaton-based specification miners. In: Proceedings of the 13th Working Conference on Reverse Engineering. IEEE Computer Society.

Lo, D., Maoz, S.,2009. Mining hierarchical scenario-based specifications. In: Proceedings of the International Conference of Automated Software Engineering. IEEE Computer Society.

Lo, D., Khoo, S.-C., Liu, C.,2007. Efficient mining of iterative patterns for software specification discovery. In: Proceedings of the SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM.

Lo, D., Khoo, S.-C., Liu, C., 2008. Mining temporal rules for software maintenance. Journal of Software Maintenance and Evolution: Research and Practice 20, 227–247.

Lorenzoli, D., Mariani, L., Pezzè, M.,2008. Automatic generation of software behavioral models. In: Proceedings of the International Conference on Software Engineering. ACM.

Lucane. www.sharewareconnection.com/lucane.htm (accessed 2011).

Manning, C., Raghavan, P., Schutze, H., 2008. Introduction to Information Retrieval. Cambridge Press.

Mariani, L., Pastore, F.,2008. Automated identification of failure causes in system logs. In: Proceedings of the International Synposium on Software Reliability Engineering. IEEE Computer Society.

Mariani, L., Pezzè, M., 2007. Dynamic detection of COTS components incompatibility. IEEE Software 24, 76–85.

Mariani, L., Pastore, F., Pezzè, M., Santoro, M., 2011a. Mining finite-state automata with annotations. In: Lo, D., Khoo, S.-C., Han, J., Liu, C. (Eds.), Mining Software Specifications, Data Mining and Knowledge Discovery. CRC Press.

Mariani, L., Pastore, F., Pezzè, M., 2011b. Dynamic analysis for diagnosing integration faults. IEEE Transactions on Software Engineering 37.

Mariani, L., Pezzè, M., Riganelli, O., Santoro, M., 2010. SEIM: static inference of interaction models. Proceedings of the 2nd International Workshop on Principles of Engineering Service Oriented Systems.

Open Hospital, sourceforge.net/projects/angal (accessed 2011).

Pradel, M., Bichsel, P., Gross, T.R.,2010. A framework for the evaluation of specification miners based on finite state machines. In: Proceedings of the International Conference on Software Maintenance. IEEE.

Raffelt, H., Steffen, B.,2006. Learnlib: a library for automata learning and experimentation. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering. Springer.

Ramanathan, M.K., Grama, A., Jagannathan, S.,2007. Path-sensitive inference of function precedence protocols. In: Proceedings of the International Conference on Software Engineering. IEEE Computer Society.

Rapid Miner. rapid-i.com (accessed 2011).

Raz, O., Koopman, P., Shaw, M.,2002. Semantic anomaly detection in online data sources. In: Proceedings of the International Conference on Software Engineering. ACM.

Reiss, S.P., Renieris, M.,2001. Encoding program executions. In: Proceedings of the International Conference on Software Engineering. IEEE Computer Society.

Safyallah, H., Sartipi, K.,2006. Dynamic analysis of software systems using execution pattern mining. In: Proceedings of the International Conference on Program Comprehension. IEEE Computer Society.

Schneider, S., Beschastnikh, I., Chernyak, S., Ernst, M., Brun, Y.,2010. Synoptic: summarizing system logs with refinement. In: Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques. USENIX Association Berkeley.

SourceForge home page. http://sourceforge.net/ (accessed 2011).

TagSoup home page. http://home.ccil.org/cowan/XML/tagsoup/ (accessed 2011).

Thingamablog. www.thingamablog.com (accessed 2011).

Thummalapenta, S., Xie, T.,2009. Alattin: mining alternative patterns for detecting neglected conditions. In: Proceedings of the International Conference of Automated Software Engineering. IEEE Computer Society.

Walkinshaw, N., Bogdanov, K.,2008. Inferring finite-state models with temporal constraints. In: Proceedings of the International Conference of Automated Software Engineering. IEEE Computer Society.

Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.,2007. Reverse engineering state machines by interactive grammar inference. In: Proceedings of the Working Conference on Reverse Engineering. IEEE.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering – An Introduction. Kluwer Academic Publishers.

Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.,2006. Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the International Conference on Software Engineering. ACM.

Zhong, H., Zhang, L., Xie, T., Mei, H.,2009. Inferring resource specifications from natural language API documentation. In: Proceedings of the International Conference of Automated Software Engineering. IEEE Computer Society.

David Lo received his PhD in computer science from the National University of Singapore and he is currently an assistant professor at the School of Information Systems, Singapore Management University. His main research area is software engineering, with specific focus on specification mining, software repository mining, debugging, testing, empirical software engineering, and software analytics leveraging information retrieval, data mining, and natural language processing techniques. He also work on various topics in data mining, with specific focus on frequent pattern mining, discriminative pattern mining, and social network mining. He regularly serves as a program committee member for several conferences related to software engineering and data mining. He has also served as an organizing committee member for several conferences/workshops.

Leonardo Mariani received his PhD in computer science from the University of Milano Bicocca and he is currently a researcher at the same university. His main research area is software engineering, with specific focus on testing, static and dynamic analysis, model inference, and self-healing technologies. He regularly serves as a program committee member for several conferences related to software engineering and actively contributes to the organization of events related to his research interests. He has worked on several national and european research projects.

Mauro Santoro received his MS and PhD in computer science from the University of Milano Bicocca. He is a postdoctoral researcher at the same University. His research interests regard software engineering and software quality assurance. In particular his main activities focus on static and dynamic analysis, model inference, and automatic test case generation. Mauro also works on the European Union FP7 project "PINCETTE".