Automated Oracles: An Empirical Study on Cost and Effectiveness

Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella
Fondazione Bruno Kessler
Trento, Italy
{cunduy,marchetto,tonella}@fbk.eu

ABSTRACT

Software testing is an effective, yet expensive, method to improve software quality. Test automation, a potential way to reduce testing cost, has received enormous research attention recently, but the so-called "oracle problem" (how to decide the PASS/FAIL outcome of a test execution) is still a major obstacle to such cost reduction. We have extensively investigated state-of-the-art works that contribute to address this problem, from areas such as specification mining and model inference. In this paper, we compare three types of automated oracles: *Data invariants*, *Temporal invariants*, and *Finite State Automata*. More specifically, we study the training cost and the false positive rate; we evaluate also their fault detection capability. Seven medium to large, industrial application subjects and real faults have been used in our empirical investigation.

Categories and Subject Descriptors

D.2 [**Software Engineering**]; D.2.5 [**Testing and Debugging**]

General Terms

Experimentation, Reliability

Keywords

Specification Mining, Automated Testing Oracles, Empirical Study

1. INTRODUCTION

Nowadays, the Internet is connecting devices, systems, and people at an unprecedented speed. Software systems have to evolve at a similar pace to deal with new use cases, new categories of users, and business changes. Taking some of today's social network systems as examples, we can see that new features are being released in a matter of weeks. New user interfaces and new supported devices are also introduced continuously. Software testing is involved in every change; hence, its cost is increasingly high. Automated techniques that are able to generate effective test cases and judge the test outcome for software testing are needed to deal with the testing cost.

The "automated oracle problem" – how to judge the behaviors of a system under test and how to decide the PASS/FAIL outcome of a test execution – is still a major obstacle to such cost reduction. In the literature, there has been a large amount of work investigating software specification mining [8]. Mined specifications can be used, on one hand, in software maintenance. They help understand software systems, including legacy systems, when specification documents are missing or outdated. On the other hand, mined specifications can be used to find software faults, by acting as guards or criteria to evaluate behaviors of a system under test (SUT). In other words, they are software oracles, which are automatically learned or mined. We name them *automated oracles*.

The principal idea underpinning automated oracles is that when a SUT is running, its execution traces are collected. From such traces (which are assumed to be non-faulty), oracles of different types are learned, and as long as the SUT operates properly, the learned oracles can be continuously refined and updated with new incoming traces. These oracles report problems if any new trace violates them. Automated oracles are particularly useful in regression testing where good oracles that are learned from a version can be used to check the execution traces of the next new version. They are also useful in systems that continuously evolve or systems that can exhibit new behaviors at runtime. In these systems, automated oracles will be learned and used for checking continuously.

We have investigated the literature including the main approaches that can be used as automated oracles and we have selected three types or oracles, based on their characteristics, popularity and supporting tools: *Finite State Automata* (FSA), *Data Invariants*, and *Temporal Invariants*. In this paper, we extensively evaluate these oracles in terms of cost and effectiveness in an industrial context where they are applied to medium and large subject applications. The key contributions of the paper are twofold:

- This is the first independent, large scale study on the cost/benefits involved in oracle learning, in terms of false positive rate, fault detection and needed resources, conducted on 7 real-world systems. To the best of our knowledge, the size of the analyzed traces (more than 10,000 events, on average) exceeds by a large amount the trace size considered in all existing empirical studies in the area of specification mining.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491434

- The empirical study provided us with valuable insight about the practical applications of automated oracles, specifically the trade-off between cost and capability to detect real faults.

The remainder of the paper is organized as follows. Section 2 gives a pragmatic background on the selected oracle types and introduces related work. Section 3 discusses in detailed the design of our study and the selected subject applications. Section 4 and 5 show the results that we obtained and discuss them. We discuss the relevant threats to the validity of our study in Section 6. Finally, Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 A Pragmatic Background

Different types of automated oracles have been proposed: *Finite State Automata* (FSA), which model the behaviors of a SUT by means of finite state machines, *Data Invariants*, which pose constraints on software variables at specific program locations, and *Temporal Invariants*, which guard the occurrence of events during the execution of the SUT according to specific patterns. All these types of oracle have received substantial research attention and supporting tools are available to infer and exploit such automated oracles [5, 7, 16, 15, 1]

While in their original proposals, some of the automated oracle inference tools that we consider have not been evaluated empirically for their support to automated fault/anomaly detection, in this work, we focus on their effectiveness when used as automated oracles for seven real world subject applications. We have selected KLFA[1], [10] for FSA oracles, Daikon[2] [6] for Data Invariants as oracles, and Synoptic[3] [2] for Temporal Invariants as oracles. These are quite popular and active tools to date. In the case of Synoptic, we use only the temporal invariant inference module, although its main feature is the generation of automata, which are however supposed to be used mainly for software understanding.

To illustrate the functionality of those tools we introduce some examples of traces of a simple shopping cart system (*SimpleCart*) and show how automated oracles can be inferred from them.

SimpleCart is a simple shopping system that sells a single subject, its user can add items to a cart, remove items from it, and check out when she is happy with the number of items. Table 1 shows 3 traces obtained from the execution of SimpleCart. The first column shows the events that were fired; the second column shows the values of the global variable *itemInCart*, that was changed after each event. The first trace contains 4 events ⟨*add, add, remove, checkout*⟩, the second trace contains 7 events ⟨*add, add, add, remove, remove, add, checkout*⟩, and the last trace contains 4 events ⟨*add, remove, add, checkout*⟩

The value of variable *itemInCart* changes as the consequence of some events: when *add* occurs, *itemInCart* is increased by 1, when *remove* occurs *itemInCart* is decreased by 1.

From the traces in Table 1 KLFA infers the finite state machine shown in Figure 1. This automaton can be used as a software oracle to check if future traces are recognized (aka accepted) by the automaton; otherwise, the system exhibits an anomalous behavior, either the manifestation of a fault or a false positive of the automated oracle.

[1]http://www.lta.disco.unimib.it/tools/klfa/
[2]http://groups.csail.mit.edu/pag/daikon/
[3]http://code.google.com/p/synoptic/

Table 1: Example of 3 traces of SimpleCart, N: itemInCart

trace 1:

Event	**N**
add	1
add	2
remove	1
checkout	1

trace 2:

Event	**N**
add	1
add	2
add	3
remove	2
remove	1
add	2
checkout	2

trace 3:

Event	**N**
add	1
remove	0
add	1
checkout	1

Table 2: Examples of inferred temporal invariants, produced by Synoptic

AlwaysFollowedBy (→)
INITIAL → add
INITIAL → checkout
INITIAL → remove
add → checkout
remove → checkout

AlwaysPrecedes (←)
add ← checkout
add ← remove
remove ← checkout

NeverFollowedBy (↛)
checkout ↛ add
checkout ↛ checkout
checkout ↛ remove

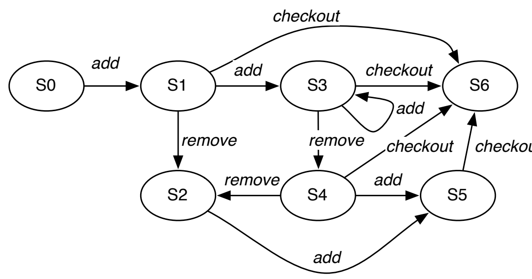

Figure 1: An example of inferred automata, produced by KLFA

Synoptic considers currently three types of temporal invariants: *AlwaysPrecedes*, *AlwaysFollowedBy*, and *NeverFollowedBy* [2]. They capture the occurrence relationship between pairs of events. In our running example, with the same input traces Synoptic infers 11 temporal invariants as shown in Table 2.

Daikon infers invariants at program locations of interest. They are often before and after method invocations and can be used as pre/post-conditions. For instance, for the event *add* (actually, the corresponding method *add* in the source code), Daikon infers that *itemInCart* is always greater than zero (*itemInCart* > 0) after method invocation. It infers the same invariant before the invocation of *checkout*.

The invariants produced by Synoptic and Daikon can be used as oracles to check the occurrence of events or the value of variables at runtime. Any violation of such invariants will raise an alarm for investigation. As with KLFA, an invariant violation may point to a software fault or may be a false positive of the automated oracle.

As a concrete example, assuming a fault in SimpleCart that allows removing items even when the cart is empty, the following new trace may be produced:

trace that exhibits the fault:

Event	**itemInCart**
remove	-1
remove	-2
add	-1
checkout	-1

All three types of oracles can detect this fault: the FSA produced by KLFA in Figure 1 rejects this trace, meaning that KLFA detects the fault; the temporal invariant *add* $\longleftarrow$ *remove* produced by Synoptic is violated by this trace because *remove* cannot occur before *add*, thus, Synoptic detects the fault; finally, the negative values of *itemInCart* also violate the invariants produced by Daikon; hence, it can also detect the fault.

2.2 Related Work

There is a large body of work on dynamic model inference and specification mining [1, 3, 7, 11, 12, 13]. The approaches described in these papers have been shown to be able to effectively detect software faults and anomalies. However, the empirical validation reported in these papers was conducted by the authors themselves, usually on small scale benchmarks that were often known to exhibit some of the problems specifically addressed by the proposed technique. Our work is the first one that compares three different types of oracle on a common benchmark, consisting of medium and large size systems, for which long execution traces are available. Since we have not been involved in the development of any of the evaluated tools, our study is an independent evaluation. We focus on the practical use of automated oracles, using the best available tools out of the box, to see the potential obstacles that a software practitioner might encounter if she plans to adopt automated oracles.

Staats et al. [14] studied the understandability of invariants when they are used as test oracles. Their results indicate that users have problems in understanding those oracles. Our study shares the same topic of automated oracles. However, we investigate two completely different aspects: false positive rate and fault detection.

In this section, we summarize the related work that shows the application of automated oracles in detecting faults and anomalies.

Mariani et al. [11] presented an approach that combines classic dynamic analysis with incremental finite state generation. The generated model can effectively detect faults and their causes of failures when it is trained by a diverse set of tests that cover well the execution space. Experiment results, obtained through a set of small size trace samples (up to 406), have demonstrated the viability and effectiveness of the approach. Moreover, the authors implemented a heuristic in their tool, BCT, to filter out false alarms. BCT is a core component of KLFA, the tool that we used in this study.

A Markov model of the system behavior is learned by means of the active learning algorithm proposed by Bowring et al. [3], but their aim is not anomaly detection and automated oracle creation. Rather, the authors propose to use the learned model for the identification of new behaviors that require the extension of the existing test plan with additional test cases.

Sekar et al. [13] proposed to focus only on system calls to learn a FSA. In fact, the approach deals with system security and aims at detecting anomalous sequences of system calls, which are likely to point to intrusion attempts and malware. Since this approach is specifically tailored for security testing, it cannot be included in our study, which instead considers general purpose oracles. The empirical validation reported by the authors [13] takes into account only the occurrence and the actual detection of security issues.

While one of the envisaged uses of Daikon is bug avoidance [6], this tool has not undergone systematic empirical evaluation along this specific dimension. A small-scale evaluation of Daikon, in the context of program changes and regression testing, is reported in another paper about Daikon [4]. The program used for this evaluation is `replace`, a small program from the Siemens test suite.

Eclat [12] is a JUnit test case generator that tries to produce candidate oracles from passing runs. The operational model used as unit test case oracle is produced by Daikon. The tool was evaluated on a set of small Java programs, the largest of which consists of less than 2K lines of code.

Diduce [7], a data invariant mining tool that set the foundations for successive developments in the field, among which Daikon [6], was originally evaluated on four Java programs. The experiences reported by the authors about the use of Diduce are very encouraging, since the tool was able to reveal interesting faults that may have gone otherwise unnoticed. The kind of empirical evaluation conducted in this work can be classified as an experience report, which is fundamental in the early stages of a new research, but should be complemented by more systematic studies, such as the one presented in this paper, when the discipline matures. Synoptic [2, 1] has also been evaluated only in small-scale studies, in the experience report category, with quite promising results, both in terms of bug finding and of increasing the developers' confidence in the implementation.

3. EXPERIMENT DESIGN

While automated oracles have the potential to reveal software faults [9, 2, 6], they come with some cons, however. The biggest issue with automated oracles is the rate of false alarms, or false positives. When such rate is intolerably high, any problem reported by automated oracles will be unreliable; hence developers will just ignore it. In this section, we describe the design of the empirical evaluation used to assess the effectiveness of the automated oracles produced by KLFA, Synoptic, and Daikon on a set of medium to large industrial subject systems. The choice of these three tools is motivated as follows: KLFA is a good representative for FSA oracles, Synoptic represents temporal invariants, and Daikon data invariants. The following are the research questions that we have investigated:

- **RQ1** [*False alarms*]: What is the rate of false positives associated with the use of automated oracles? Which one triggers more false alarms?
- **RQ2** [*Fault detection*]: Once trained, can automated oracles reveal new faults occurring in running applications? Which one is more effective?
- **RQ3** [*Training and checking cost*]: How long does it take for training? Once oracles are trained, how much time does it take for checking a new trace?

We adopt a dynamic analysis assessment procedure for each subject, simulating the real use of the automated oracles: A subject system **P** is running, its execution is monitored to obtained traces, and different automated oracles are inferred from those traces. Then, due to new usages (new traces), the automated oracles may report alarms when the execution violates them. Alarms might due to a fault that has been triggered, or they may be wrong (false positives). Figure 2 illustrates this procedure. The training traces in **T** are used to infer oracles $\{O_1, O_2, \dots, O_N\}$ of different types (in our case, $N = 3$, the oracles being FSA, temporal invariants, or data invariants). Given the new traces in **F**, the inferred oracles may raise alarms, which could be true faults or false positives.

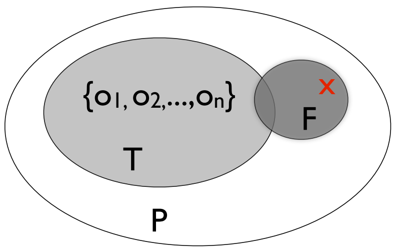

Figure 2: Oracle learning and fault reporting

The subject systems used in our study are shown in Table 3. They are Java applications from different domains and of different sizes, up to 94,550 NCLoCs (Non-Comment Lines of Code). They have been selected so as to maximize diversity of features and representativeness of relevant application domains. All applications come with a large manual JUnit test suite, up to more than 4,600 test cases (we consider each JUnit test method as one test case). We execute these test cases to get traces from the subjects. Each test case produces one trace if the test succeeds; we discard failed tests (actually, there was only a small number of failed tests from all the subjects and failures were due to concurrency issues, which cannot be controlled from the test execution environment).

During test execution, Daikon was used to obtain the traces, because the Daikon trace format is general enough to be compatible with the input required by the other two tools, KLFA and Synoptic. The Daikon trace format consists of program point definitions (*i. e.*, identifier, variable names and types at executed program locations) and the concrete values of the variables during test execution. Daikon traces can be used directly by Daikon, and from such traces we extract sequences of method invocations (ENTER) and method exits (EXIT) in the exact order as they occur during test execution. These sequences are the input traces for Synoptic and KLFA; we name these traces as *sequence traces*.

Sequence traces are often very long (the average length of the sequences for our subjects is 10,689), and each entry in a sequence is often a full Java method signature including package name, class name, method name, and then the parameters' long signatures. This slows down the performance of Synoptic and KLFA and requires more RAM during trace processing. Therefore we developed a trace compression algorithm that replaces sequence entry values with their hash keys, consisting usually of just 1 or 2 characters. The algorithm takes into account the occurrence frequency of sequence entries to assign shorter hash keys for entries that are most frequent. As a result, the size in bytes of the traces is reduced dramatically, up to 96%, from 5.9Gb to 224Mb. This helps Synoptic and KLFA deal with more and with longer traces.

3.1 RQ1: False positives

To study the false positive rate we use only the traces in the training set T, under the assumption that all of them are the result of normal, accepted behaviors of **P**. We sample N traces from **T** and divide the sampled set into two disjoint sub sets, T_1, and T_2, containing 90% and 10% of the traces in **T**, accordingly. T_1 is used to infer automate oracles, while T_2 is used for cross validation. Similar to the 10-folds cross validation technique in machine learning, we repeat this process to change the selection of traces for training and validation 10 times so that every trace in T will appear at least once in T_2.

For each automated type of oracle (FSA, temporal, and data oracles), for each sample set T of size N of training traces, the size of the sub set T_2 is $N_2 = N/10$, and the false positive rate is calculated as follows:

$$F_{positive} = \frac{number\ of\ rejected\ traces\ from\ T_2}{N_2}$$

A trace from T_2 is rejected if the automated oracles learned from T_1 do not accept it. In other words, the learned oracles report an alarm when processing the trace. This is a false alarm because all the traces in T are assumed to be fault-free ones, hence no alarm should be raised.

3.2 RQ2: Fault detection

Investigation of the fault detection capability of automated oracles requires substantial manual work. In fact, faults reported in bug tracking systems and later fixed in the code have to be manually re-injected in the code under analysis. For this reason, we focus this part of our empirical investigation on a single subject, **commons-collections**. **Commons-collections** has a rich and large set of test cases. Such a set of tests generates many traces that can be used for learning the automated oracles. The key factor for this choice is that the Apache Software Foundation[4] has a very good issue tracking system where we could find detailed information about faults: at which version a fault is detected and by which tests. We have selected 7 real faults of **commons-collections** for this study. More details about them are provided in Section 4.2.

To evaluate the fault detection capability of an oracle type, we work with one single real fault at a time. We consider one test case that reveals the fault; this test produces a faulty trace. All other (non-failing) tests are used to produce training traces to infer the oracles. For each oracle type, we say that it might detect the fault (candidate true positive) if at least one of its learned invariants rejects the faulty trace (for temporal and data oracles) or the trace is not accepted by the FSA (for FSA oracle). To confirm that an oracle actually detects a fault (true positive), we manually inspect the output reports produced by the oracle to find if there are any direct link between the reports (i.e., the violated invariants or the path taken in the FSA) and the fault.

[4]`http://www.apache.org`

Table 3: Subject systems used in our study: their description, size in terms of number of non-comment lines of code (NCLoc), and their test suite size

Name	Size (NCLoC)	Test Size	Description
commons-collections	17,590	4,601	Apache collection library for Java
commons-math3	50,924	2,392	Apache mathematic library for Java
xstream	17,202	967	Library for serializing objects to XML and back
jgap	43,501	1,415	Genetic programming library
openmrs-api	78,381	2,400	Core API of OpenMRS, a medical record system
jfreechart	94,550	2,218	Library to create charts in Java
joda-time	27,213	3,899	Date and time library

3.3 RQ3: Training and checking cost

Training and checking cost is measured as the amount of execution time (real time) each tool needs to process respectively all the training traces or the trace to be checked. Since during the evaluation of **RQ1** (false positive rate) we found a linear relationship between the number of traces processed and the amount of time needed to process them, to answer **RQ3** we only need to estimate the per-unit processing time of each tool. Specifically, we measure the time needed to process a unit consisting of 100 traces.

4. RESULTS

In this section, we present the results obtained from running Daikon, Synoptic, and KLFA on 7 application subjects. We traced only the application code, excluding the libraries that the applications use. For Daikon and Synoptic, we use their default configuration options, that have been fine-tuned in previous studies. For KLFA we limit its minimization step to 50 (*-minimizationLimit 50*), because otherwise KLFA cannot finish its processing even on a small number of traces. This is one suggestion of KLFA's authors to increase KLFA's performance and the parameter does not affect the generalization of KLFA output models. This means that it does not influence the acceptability of the models with respect to new traces. In other words, it does not affect the results of KLFA in our study.

4.1 False positive rate

4.1.1 Daikon - Data Invariants

Figure 3 depicts the box-plot of false positive rates of Daikon on one of our subjects, **openmrs-api**, with respect to different training sample sizes. For a sample of 1000 traces, 10 runs are performed. In each run, 900 traces (90%) are used for training the oracles and 100 traces (10%) are used for cross validation. The false positive rate is, then, calculated based on how many traces among the cross validation traces are rejected by the learned oracles. The data in the figure show that the *false positive rate reduces when the size of the training sample increases.* When the sample size is 100, the false positive rate falls in the range of 40% to 50%, while at the sample size of 1500, the rate is in between 10% to 20%. Figure 3 shows also the average false positive rate, as the line that connects the average values at different sample sizes. We also see that on average the false positive rate reduces when the sample size increases.

In Figure 4, we put together the average false positive rate of Daikon on all 7 studied subjects. We ran the experi-

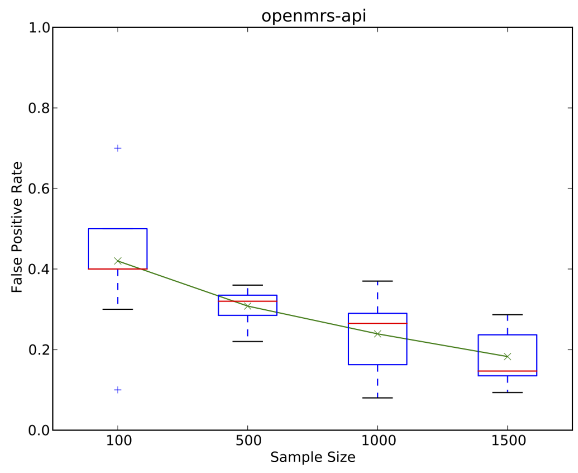

Figure 3: False positive rate of Daikon on openmrs-api

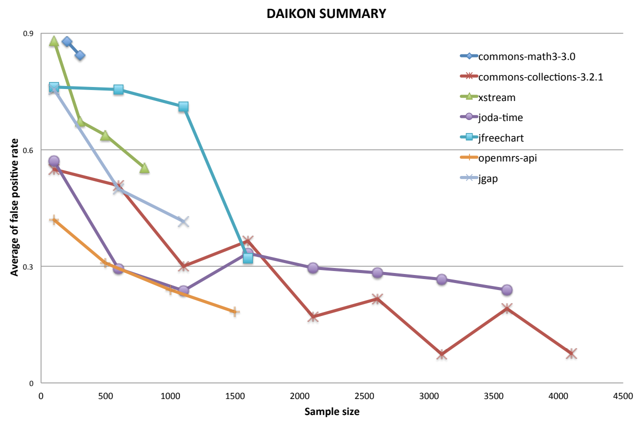

Figure 4: Average false positive rate of Daikon on the subjects

ments based on the traces available from the subjects and for each subject the sample size was increased from 100 to the maximum number of traces that the subject has. For example, **jfreechart** was run from 100 to 1600 traces, while **commons-collections** was run from 100 to 4100 traces. Overall, with Daikon we observe the decrease of false positives when the size of the training set increases. This means that the more we train the data invariants, the less they overfit the training data. Hence, they better generalize to new data, accepting more legal behaviors. As a result, the false positive rate is reduced. The decrease of the false positive rate is not monotonic for 2 subjects, **joda-time**, and **commons-collections**. We speculate that this is specifically due to trace sampling: particular sample sizes induce a sample in which cross-validation traces are similar to the training traces. Hence, the former are likely accepted by the learned oracles. As a result, the false positive rate reduces quicker than usual at such sample sizes.

Regarding the rate of false positive with respect to the training sample size, as depicted in the figure, we can notice that when the training sample size is smaller than 1000 traces, for most of the subjects the false positive rate is higher than 30%, which is a relatively quite high rate. The best and smallest false positive rate obtained with Daikon is 7.48%, with **commons-collections** with the highest sample size of 4100.

4.1.2 *Synoptic - Temporal Invariants*

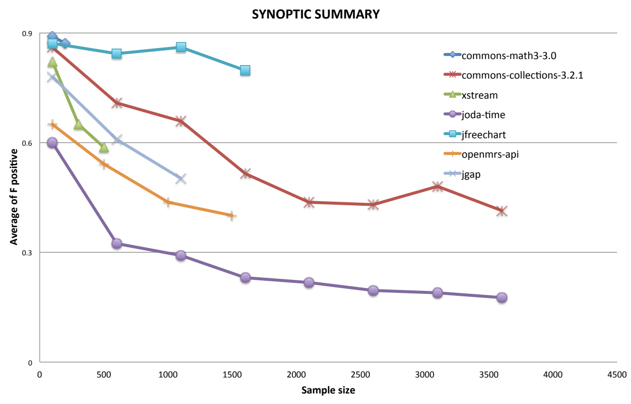

Figure 5: The average of false positive rate of Synoptic on the subjects

Figure 5 puts together the average false positive rate of Synoptic on the 7 application subjects. Here, on all subjects we also observe that the average false positive rate reduces when the sample size increases. This means that the more we train the temporal invariants, the smaller false positive rate they report.

Overall, when the sample size is smaller than 1500 on all cases except **joda-time** the false positive rate is greater than 40%, higher than the rate obtained by Daikon (even with 1000 traces, Daikon's rate is 30%). With **joda-time**, Synoptic is able to lower the false positive rate to less than 30% when the sample size is greater than 1100. The best rate that Synoptic achieved is 17.53% with **joda-time** when the sample size is 3600. **Commons-collections** has more traces, but we could not increase the sample size due to the associated training time, which becomes unmanageable (even on a cluster). We discuss this in Section 4.3.

4.1.3 *KLFA - Finite State Automata*

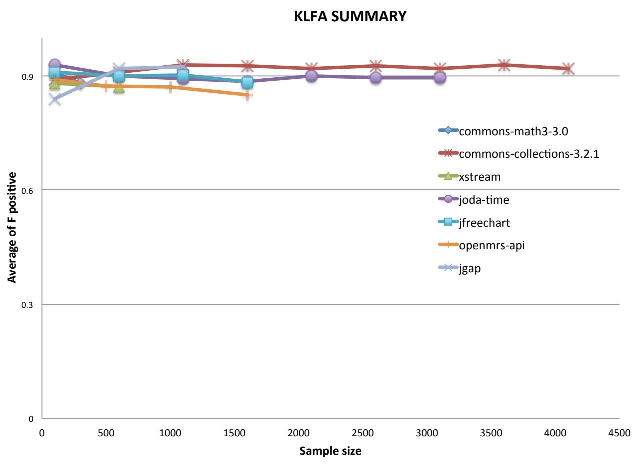

Figure 6: The average of false positive rate of KLFA on the subject

Among the three tools, KLFA is the worst performer in terms of false positive rate (see Figure 6). In all cases, it can only slightly reduce the rate when we increase the sample size. KLFA performs the best with **openmrs-api**, where it is able to reduce the false positive rate from 89% to 85.05%, which is still too high to be used in practice.

The reason behind this poor performance is that we considered system-level traces that are quite long (their average length is 10,689 events/trace). When checking a new trace, the automaton inferred by KLFA has to accept every single event in the exact sequential order of occurrence for an entire system-level trace. This is quite unlikely to happen at the system-level, while results may be substantially different at the component/unit level, where shorter traces are produced. Our recommendation for this kind of oracle is to use it within a small scope, such as a single unit or component. The false positive rate is expected to be much smaller in such a scope.

4.2 Fault detection

In this study we selected 7 real faults of **commons-collections** from its issue repository:
`https://issues.apache.org/jira/browse/COLLECTIONS`.
Table 4 shows the IDs of the faults and their descriptions. Detailed information for each fault is available from `https://issues.apache.org/jira/browse/[FaultID]`. These faults were selected based on their diversity and severity. They are all major faults and they range from missing exception handling to logical faults, like wrong method invocation COLLECTIONS-219 or wrong implementation COLLECTIONS-359.

We ran our experiments on each fault individually, so that in each run only one single fault is present. Each fault is revealed by one unique test case. The fault will manifest itself in the corresponding faulty trace when its revealing test case is executed. Because of the faults, the faulty traces might contain some sequences of events or data values that violate the learned oracles. All other (non-faulty) traces obtained from PASS tests are used for oracle training, because – as the results on the false positive rate indicate – the more we train the oracles, the less false positive rate we obtain in most cases.

Table 4: **Fault descriptions**

Fault ID	Description
COLLECTIONS-411	IndexOutOfBoundsException in ListOrderedMap.putAll(int index, Map<? extends K, ? extends V> map).
COLLECTIONS-219	The CollectionUtils.removeAll method calls the ListUtils.retainAll method instead of the ListUtils.removeAll method.
COLLECTIONS-400	Missing a NullPointerException check.
COLLECTIONS-438	Class ExtendedProperties has problem with a key composing of only space character.
COLLECTIONS-299	Method convertProperties of the class ExtendedProperties loses non-String values.
COLLECTIONS-359	When duplicates are present in a list, ListUtils.intersection, the intersection of two lists should give the same result regardless of which list comes first.
COLLECTIONS-331	NullPointerException at the CollatingIterator class when Comparator is null.

Table 5: Fault detection results by Daikon (Data Invariants), Synoptic (Temporal Invariants), and KLFA (FSA). Each tool has two corresponding columns: Reported (R) = 1 if the tool reports at least an alarm, and True Positive (TP) = 1 if the reported alarms are verified to point to the corresponding fault.

Fault ID	Daikon		Synoptic		KLFA	
	R	TP	R	TP	R	TP
COLLECTIONS-411	1	0	1	0	1	0
COLLECTIONS-219	0	0	1	1	1	1
COLLECTIONS-400	1	1	0	0	1	0
COLLECTIONS-438	1	0	0	0	1	0
COLLECTIONS-299	0	0	1	1	1	1
COLLECTIONS-359	0	0	1	0	1	0
COLLECTIONS-331	1	0	1	1	1	0
Total	4/7	1/7	5/7	3/7	7/7	2/7

Table 5 summarizes the fault detection results of Daikon (Data Invariants), Synoptic (Temporal Invariants), and KLFA (FSA). As expected, given its very high false positive rate, KLFA raises alarms on all the faults, meaning that its learned oracles report problems with respect to all the faulty traces (column KLFA/R). Synoptic and Daikon also report many faults. Daikon reports 4 out of 7 faults (column Daikon/R), while Synoptic reports 5 out of 7 (column Synoptic/R).

We investigated the fault reports provided by the tools to see if the invariant/FSA violations are actually due to to the faults. In other words, we manually verified whether the reported faults are truly detected by the automated oracles. The second, forth, and last columns of Table 5 show the faults that were truly revealed by Daikon, Synoptic, and KLFA. Daikon revealed only 1 fault, while Synoptic revealed 3 and KLFA revealed 2 faults. On one hand, this shows that automated oracles can, indeed, detect faults. On the other hand, what is also very interesting is that Daikon and Synoptic result to be complementary of each other: all the true positives are exclusively detected by either Daikon only or by Synoptic only. This finding suggests that in practice, we should combine these two types of oracle to maximize the probability of detecting faults. In addition, the set of faults revealed by Synoptic subsumes those detected by KLFA. This can be explained by the fact that both tools depend on the temporal occurrence of events in sequential traces. In practice, both Daikon and Synoptic should be used to maximize the fault detection capability of automated oracles, while, according to our empirical results, KLFA does not need to be used if Synoptic is also used.

In what follows, we give some examples of fault reports and how they relate to the corresponding faults.

COLLECTIONS-400 Daikon report:

At ppt org.apache.commons.collections.CollectionUtils-.addIgnoreNull(java.util.Collection, java.lang.Object):::ENTER, Invariant **'collection != null'** invalidated by sample collection=null

This is a Daikon report saying that at the entry of the method *addIgnoreNull*, Daikon has learned an invariant stating that the *collection* parameter must be non-null. However, when executing the fault revealing test *org.apache.commons-.collections.TestCollectionUtils.addIgnoreNullBug400*, a null pointer has been observed, which violates the invariant. Such null pointer is a consequence of the bug, hence, in this case, Daikon's automated oracle violation is a direct manifestation of the bug.

COLLECTIONS-219 Synoptic report:

Invariant: **24R AlwaysFollowedBy(t) 1UA** invalidated by this counter example: [INITIAL] [QH] [QF] [5X] [5Y] [QJ] [QK] [26T] [1UA] [24R] [2T8] [TERMINAL]

Events are encoded as follows:

1UA:	org.apache.commons.collections.ListUtils-.retainAll(java.util.Collection, java.util-.Collection):::ENTER
24R:	org.apache.commons.collections.ListUtils-.retainAll(java.util.Collection, java.util-.Collection):::EXIT253
2T8:	org.apache.commons.collections-.CollectionUtils.removeAll(java.util-.Collection, java.util.Collection):::EXIT1251

From the report we learn that Synoptic has inferred this invariant, **24R AlwaysFollowedBy(t) 1UA**, meaning that the event *org.apache.commons.collections.ListUtils.retainAll(java.util.Collection, java.util.Collection):::EXIT253* is always followed by a *org.apache.commons.collections.ListUtils-.retainAll(java.util.Collection, java.util.Collection):::ENTER*. However, from the counter example trace, after the very last event 24R there is no 1UA. Thus, the invariant is violated. This is because of a fault in the code, that calls *retainAll* instead of *removeAll*. This happens in the body of *org.apache-.commons.collections.CollectionUtils.removeAll(...)*. As a consequence of this fault, the output event sequence changes and the last three events, which violate the invariant, and their order [1UA] [24R] [2T8] point directly to the fault.

The same fault *COLLECTIONS-219* is revealed also by KLFA, after inferring a model and checking the same counter example. In fact, KLFA expects to see either a branch going to 1UA, or another one going to 2W8, from 2T8, according to the learned model. However, 2T8 is the last event in the trace and no final state is reached in the KLFA model. Hence, KLFA rejects the trace and reveals a true fault.

4.3 Training and checking cost

The input traces for Synoptic and KLFA are sequences of events. Their size is measured in terms of the number of events. Since these traces are composed of method invocations and exits extracted from Daikon traces, the size of the corresponding Daikon traces can be standardized to the same measurement as the number of events (or Daikon data points, in Daikon's language). Among the subject applications, **xstream** produces very long traces; the average size of an **xstream** trace is 65,400 events, much higher than the average trace size of all other subjects – 3,900 events. Hence, the cost to process **xstreams**' traces is of a different category. We decided to exclude **xstream** from this part of the study. In fact, in this regard **xstream** is an outlier, not meaningful when estimating the average training and checking cost.

Table 6 shows the average clock time needed for Daikon, Synoptic, and KLFA to learn oracles for a unit of 100 traces and 3900 events on average. These data were measured on a cluster system in which the CPU speed of each node is 2.2Ghz and the amount of RAM available is up to 256Gb. The cluster system has a high-speed I/O storage, therefore the data shown in the table might be smaller than the actual amount of time needed on a hardware not equipped with high speed I/O storage (in fact, trace processing involves substantial I/O). As we can observe from the data, the cost is higher for KLFA and Daikon, from 1.32 to 1.89 minutes for a set of 100 traces. From the values, we can infer that the amount of time needed to process 4000 traces, as the case of **commons-collections**, will exceed 1 hour. For KLFA and Daikon we also observed that the amount of time for checking a single trace is very small, compared to the time required for training the oracles.

Table 6: Average wall-clock time (in minutes) to complete the training and checking of a unit of 100 traces

Tool	Time (minutes)
Daikon	1.89
Synoptic	0.18
KLFA	1.32

The training cost is smaller for Synoptic, 0.18 minutes for processing 100 traces, thanks to its parallel execution. Although Synoptic learns a huge amount of temporal invariants (e.g. more than 300 thousand invariants are learned from 100 traces of **openmrs-api**), considering the relationships among all events appearing in all traces, it partitions learned invariants into groups and it runs invariant checking in parallel for new traces, up to 6 parallel processes. As a result, the total amount of time taken by Synoptic is dramatically reduced.

In terms of the amount of memory required by the tools, on average KLFA and Daikon require up to 8Gb of RAM to be able to process all the traces. Synoptic is, again, an exception: it requires about 26Gb of RAM to process all the traces.

5. DISCUSSION

We organize the discussion of the experimental results along the following dimensions: practical usefulness, adoption costs and barriers, implications for the research in the field.

5.1 Practical benefits

Automated oracles have a moderate fault detection capability (see Table 5). This represents potentially a substantial practical benefit, because these oracles are inferred without any manual intervention aimed at refining or fine tuning them. They have been used out of the box, as reported by the tools, which means developers can expose some faults (not all of them) even without manually specifying any oracle. The downside of this positive result is that the false positive rate of these tools is quite high. Even excluding KLFA, which exhibits an intolerably high false positive rate, the other two tools, Daikon and Synoptic, are still around 30% (see Figure 4), which means that on a test suite consisting of 100 test cases, developers would have to manually classify (on average) 30 FAIL outcomes as false alarms. Each such assessment might require substantial effort, because the developer has to carefully examine the execution and the violated invariant to decide whether a real fault was exposed or the inferred invariant is wrong. Since invariants are produced automatically, they are not easy to understand and validate.

Whether the balance between false positive rate and fault detection capability is acceptable or not for practical purposes is quite difficult to assess. In practice, one can expect that a test suite will expose few faults. Still, as discussed above, it would require substantial effort to recognize the false alarms. A quite representative example could be one where the test suites consist of 100 test cases, 30 of which raise false alarms. We may also assume that (at most) 1 test case exposes a real fault. For the developer, the cost-benefit balance is between manually assessing 31 failed test cases and manually tagging 30 of them as false alarms due to incorrect invariants, as compared to manually defining an oracle capable of exposing the automatically detected fault. Both processes are complex. Manual examination of the false alarms involves deep understanding of automatically generated oracles – a difficult task. Manual definition of a fault-exposing oracle might be easy or difficult depending on the kind of fault. In our example, the ratio between the two costs (manual oracle definition vs. failed test case assessment) should be 31 or higher to justify automated oracles.

In the general case, the cost for the manual assessment of the failed test cases grows linearly with the test suite size (being on average 30% of the number of test cases). On the other hand, the cost of manual oracle definition grows with the complexity and size of the program. In turn, a larger and more complex program requires larger test suites for testing. Assuming a linear proportion and taking our example as representative, we can state that on average a cost ratio above 30 between manual oracle definition and failed test case assessment may justify the adoption of automated oracles. However, project specific estimates should be carried out to make any practical decision.

Another factor to consider about the practical benefits is that automated oracles are not going to expose all faults (see Table 5) and that they are complementary in fault detection. This means that multiple oracle inference tools should be used, hence increasing the cost for the manual assessment of the false positives, and that some manual oracle definition would be still required. As part of our future work we intend to investigate whether manual oracles are complementary or overlap with automated oracles. In fact, if automated oracles were capable to detect faults that tend to go unnoticed when manual oracles are defined, the benefits of automated oracles would be strengthened and the associated costs might be regarded as more acceptable.

5.2 Adoptions costs

Adoption costs are mostly associated with the training phase, which requires substantial machine time and resources. We used a cluster to train the automated oracles and the training time was non-negligible. It involved substantial computational resources. However, we think that nowadays such resources can be obtained at reasonable costs and that such costs do not represent the major barrier to adoption, provided the practical benefits discussed above are delivered.

Additional adoption costs are associated with the generation of traces from the test case executions and with the configuration of the tools. Tracing was easily achieved thanks to the functionalities provided by Daikon. Tool configuration was also relatively easy. A few interactions with the tool developers were necessary. The considered tools are research prototypes, not commercial tools, which means they tend to be less engineered for easy of use and robustness. On the other hand, we could use them without any major problem. These tools work for Java programs. When different programming languages are targeted, the adoption barrier might become insurmountable, since, to the best of our knowledge, no porting to programming languages other than Java is available.

5.3 Future research

Although an informed decision on the practical benefits would require a careful estimation of the involved costs, the high false positive rates of the existing automated oracles make them cost effective only when manual oracle definition costs are very high (more than 30 times higher, in our rough estimate), as compared to the manual assessment of a failed test case. In practice, this might prevent any industrial adoption of automated oracles, unless their false positive rate is dramatically reduced. We think a major research direction for this area should target the reduction of the false positive rate, while preserving the fault detection capability of the automated oracles. A fault detection rate of 1% or less, at equal fault detection rate, would make the cost-benefit balance more favorable for the automated oracles in many practical cases.

This paper represent the first attempt to empirically validate automatically inferred oracles in an unbiased way, by adopting the standard procedures used in machine learning and empirical studies, and by selecting the subjects, their test cases and the faults to be revealed, without any influence from the techniques being validated. We hope that the research community will use this experiment as the baseline for further experimentation, aiming at acquiring a comprehensive body of knowledge on automated oracles and their practical effectiveness. We make our experimental material (subjects, test cases and faults) publicly available here `http://se.fbk.eu/dnguyen/public/fse2013` for replication and for further experimentation with other tools.

6. THREATS TO VALIDITY

The main threats to the validity of this experiment belong to the internal, construct and external validity threat categories.

Internal validity threats concern external factors that may affect the independent variable. Since the authors of this paper have not been involved in the development of any of the tools evaluated in the study, there is no bias in favor or against any of the three considered tools. In contrast with the empirical evaluations published in the respective papers, associated with the tools evaluated in our study, this is the first empirical investigation conducted by a third party, which is completely unrelated with the development of the tools under evaluation. This greatly increases the internal validity of the study reported in this paper, as compared to the self-evaluations available in the literature.

Another threat to the internal validity concerns the configuration of the three tools under evaluation. Whenever possible, we adopted the default configurations or the configurations suggested in the papers describing the tools and their validation. Occasionally, we also contacted the tools' authors, asking for suggestions about specific configurations. While a deeper knowledge of the tools might have produced a better tuned configuration, to minimize this threat to validity we executed a set of preliminary runs under different configurations and we took advantage of all available hints and suggestions to ensure an optimal tool configuration, before executing the actual experiment.

Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measured the performance of the three tools under evaluation. To estimate the false positive rate, we adopted the cross-validation procedure, a pretty standard way for assessing the performance after training in machine learning. The fault detection rate was measured by carefully selecting real faults reported in issue tracking systems and by manually re-injecting them into the code. We put our best effort in carrying out this task so as to ensure that each bug was faithfully reproduced in the manually mutated code. For the cost measurement, we considered the wall-clock time measured directly on the cluster where the training process was executed. While this might expose us to some variability intrinsic with process execution on cluster nodes, the actually measured times are at a granularity (minutes) that make such fluctuations (the order of seconds) not very important.

External validity concerns the generalization of the findings. We considered 7 subjects, 7 real faults of one subject, and 3 automated oracle mining tools. While we tried to select the subjects of our experiment so as to ensure maximum diversity and representativeness, further replication of the study on additional subjects is essential to be able to generalize our findings and to reduce the external validity threat.

Generalization on tools different from the three considered in this study should be also done with care. While tools in the same categories (data invariant inference, temporal invariant inference and FSA model inference) might produce similar results, it is definitely not possible to extrapolate our findings to tools that fall in other categories. The problem of extending our empirical evaluation to other automated oracle mining tools is that often such tools are just described in papers, but their authors do not make them available for third party experimentation.

7. CONCLUSION

Several attempts to address the oracle problem in software testing have been recently proposed in the literature. Automated oracles are inferred from execution traces, by mining data invariants, temporal invariants or FSA behavioral models. The work presented in this paper is the first large-scale empirical investigation of the effectiveness of automated oracles, conducted by a third party, not involved in any of the automated oracle tools being evaluated.

Results show that automated oracles can detect several real faults, but such fault detection capability comes at the price of a quite high false positive rate (30% on average). The high false positive rate makes the balance between practical benefits (revealed faults) and costs for the manual assessment of the false alarms unclear. In our (rough) estimates, a software project where manual assessment of a false alarm incurs a cost which is 30 or more times higher than the cost for manual oracle definition could benefit from the proposed approach. However, a detailed estimate should be carried out, on a project specific basis, taking into account several factors (discussed in our paper), among which: (1) the faults that would be anyway missed by automated oracles; (2) the need for multiple kinds of automated oracles; and, (3) the degree of complementarity between manual and automated oracles. The current performance of available automated oracle learning tools do not support any easy or obvious decision about tools adoption, despite their overall acceptable adoption barrier (only for Java programs), associated with the computational resources and the configuration effort involved.

Key to the success of automated oracle mining is the capability to infer oracles that exhibit a substantially lower false positive rate, as compared to the available, state of the art tools, while keeping the same or offering an increased fault detection capability. This demands for a major advancement of the research in the area. Our future research will be devoted to further empirical investigation of the effectiveness of automated oracles, to corroborate our findings and to increase their external validity. We plan also to compare manually defined oracles with automated oracles, so as to have a direct measurement of the involved costs and to determine whether they target complementary or overlapping classes of faults. Specifically, we intend to conduct a study in which the ratio between the cost for manually defining the oracle and the cost for manually classifying a failed test case as a false alarm or a real fault are measured directly.

8. ACKNOWLEDGEMENT

This work has been funded by the European Union FP7 project FITTEST (grant agreement n. 257574).

9. REFERENCES

[1] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining temporal invariants from partially ordered logs. In *Workshop on Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML '11)*, Cascais, Portugal, Oct. 2011.

[2] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 267–277, Szeged, Hungary, September 7–9, 2011.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. *SIGSOFT Softw. Eng. Notes*, 29:195–205, July 2004.

[4] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99 –123, feb 2001.

[5] M. D. Ernst. *Dynamically Discovering Likely Program Invariants.* Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[7] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[8] D. Lo, S.-C. Khoo, and J. Han, editors. *Mining Software Specifications: Methodologies and Applications.* CRC Press, 2012/09/21 2011.

[9] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[10] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117 –126, nov. 2008.

[11] L. Mariani, F. Pastore, and M. Pezzè. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Trans. Software Eng.*, 37(4):486–508, 2011.

[12] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *In 19th European Conference Object-Oriented Programming*, pages 504–527, 2005.

[13] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[14] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 188–198, New York, NY, USA, 2012. ACM.

[15] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[16] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.