Dynamic Specification Mining with Out-of-Scope Effect Awareness and Result Classification

Ziyad Alsaeed
Computer Science Dept.
University of Oregon
Eugene, OR 97403-1202
zalsaeed@cs.uoregon.edu

ABSTRACT

Dynamic specification mining techniques attempt to fill gaps in missing or decaying documentation of software systems to support software maintenance tasks such as testing or bug fixing. Current dynamic mining techniques are blind to common coding styles, and in particular to design patterns that involve dynamic data structures such as lists of listeners for event notification. Because they cannot recover properties involving these dynamic structures, they may produce incomplete or misleading specifications (e.g., suggesting a method may be “pure” because its effects are produced indirectly through event notification). We have devised an extension to current dynamic specification mining techniques that ameliorates this shortcoming. The key insight is to monitor not only values dynamically, but also properties to track dynamic data structures commonly used in design patterns. We have implemented this approach as an extension to the instrumentation component of Daikon, the leading example of dynamic invariant mining in the research literature. We have applied our tool to widely used software systems published on GitHub to illustrate and evaluate the usefulness of this pliable monitoring for elucidating the overall behavior of target systems.

KEYWORDS

dynamic analysis, specification mining, dynamic data structure analysis, design patterns

1 INTRODUCTION

So-called specification miners and invariant detectors extract potentially useful conjectures [1] about program behavior from program source code [22, 24, 25], dynamic monitoring of behavior [6, 7, 9, 10, 12, 13, 15, 16, 18, 19, 21, 23, 29], or both [8]. Ideally these would be true specifications or design descriptions, as might be produced by a human programmer with a deep understanding of a program’s design. In practice, it is difficult to infer complex relationships that would be most useful in maintaining, extending, or evaluating a software system. Current software invariant detectors are best at detecting simple relations in static structures, such as relations among simple variables stored as fields of an object or local variables of a method. Such behavior describes objects in isolation from others, but fails to capture complete object’s interactions with other peer objects in the system. More specifically, they are limited in detecting relations in dynamic data structures, including common design patterns such dynamically attached observers or views.

[1]The mined specifications are called "conjectures" or "likely-invariants" because their observation is highly dependent on the given unit tests. It is generally impossible for given tests to explore all possible paths in a system.

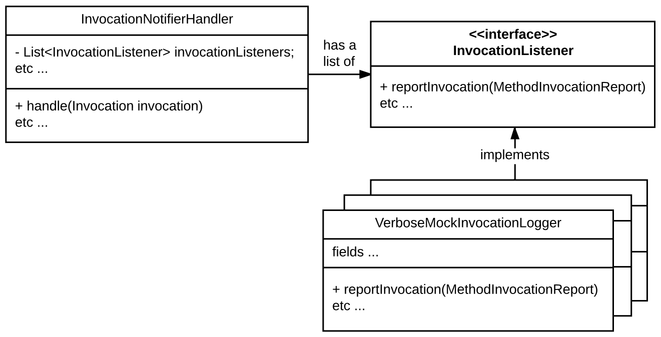
InvocationNotifierHandler
- List<InvocationListener> invocationListeners; etc ...
+ handle(Invocation invocation) etc ...
has a list of
<<interface>> InvocationListener
+ reportInvocation(MethodInvocationReport) etc ...
implements
VerboseMockInvocationLogger
fields ...
+ reportInvocation(MethodInvocationReport) etc ...

Figure 1: UML class diagram of listeners dynamically attached to an object in the Mockito framework [4].

We posit that dynamic invariant detectors can be extended to recognize important relations among objects in dynamically established relations, such as between an observer and its subject in the *observer* pattern or a view and model in the *model-view-controller* pattern. To do so requires characterizing dynamic data structures (e.g. the list of observer objects to be notified by a subject object in the *observer* pattern) and tracking relations that span variables among multiple objects. For example, the method `handle(Invocation)` from [4] (see Figure 1) loops through existing elements of its list of listeners (`invocationListeners`) and calls the method `reportInvocation(MethodInvocationReport)` on each listener which on return alters some variables for each listener. However, based on invariants reported by Daikon [12], one might believe the method has no effect (*pure*). Such misleading inference is not caused by a shortage in the given unit tests, rather by scope limitation.

The given tests to drive a system observation could heavily influence the accuracy of the specification miner. Nevertheless, established work to observe a target application symbolically (e.g. Csallner et al. [10]) suggested a notable improvement in the inference accuracy. However, such solution does not solve the issue of describing the system as a whole because of complex relationship among its objects. So, the actual challenge is to redesign the instrumentation methodologies to establish pragmatic limits of data structure exploration that are sufficient to recognize typical relations without exploding the cost of monitoring or producing excessive noise in inferred invariants.

We have designed and implemented a proof-of-principle prototype instrumentation tool called eChicory (abbreviation of Enhanced Chicory) for inferring invariants about objects in dynamic relationships by extending the front end of Daikon (Chicory) dynamic invariant detector for Java [12]. With eChicory, we demonstrate that our approach can recognize dynamic relationships that are beyond the scope of a single class or object, including relationships that arise in common design patterns such as the *observer* and *model-view-controller* patterns mentioned above. We also describe limitations of our current prototype, distinguishing between some that we believe could be straightforwardly overcome with more time and engineering effort and others that we pose as open problems for further research.

We also look at the results of the dynamic inference techniques from an unusually embraced perspective for evaluation. Because we are targeting larger test subjects compared to existing techniques, and since most of them do not have associated up-to-date specifications (ground truth), we use existing static analysis techniques to produce a partial ground truth for comparison. Method *purity* (absence of side effects) is one such partial ground truth. Instead of heavily depending on humans manually producing ground truths about target applications, then measure the techniques' precision and recall, we simply measure weather a technique has captured any sort of effect of a method or not.

We demonstrate our contribution in the context of the Java library and the Daikon [12] invariant inference tool. Similar observations, nevertheless, are applicable on different programming languages to inference tools.

2 BACKGROUND AND EXAMPLE

This section provides necessary background for discussing current specification miners' shortcomings in observing complex relationships of objects and establishes background for our proposed solution. Section 2.1 reviews existing approaches potentials and limitations and helps narrowing down the focus on the root cause of the problem and the scope of our contribution. Section 2.2 establishes an example that would help in understanding the limitations of current specification miners, that would be also used later to demonstrate out contribution. Section 2.3 shows limitation on results based on the evaluation of current techniques and how it is incapable of showing the general behavior of complex system.

2.1 Potential and Limitations of Established Specification Miners

There are many established techniques to mine software specifications dynamically. To some level, regardless of what is the source for inference and what are they looking for, these techniques should ensure software specification being up-to-date and accurate. Many dynamic inference techniques use the system's source code [8–10, 12, 13, 15, 16, 18, 21, 29]. Others infer specifications from delivered binaries [19, 23]. A few mine logs [6, 7]. These techniques differ in terms of what are they looking for in a target application. Daikon [12] and few others [9, 10, 15, 18, 29] attempt to generate specifications in the form of likely invariants at each method entrance and exit points. Other techniques [6, 7, 21] attempt to construct state transition models that capture the target system method calling sequences. These techniques intend to infer specification of objects in isolation from the other objects in the application.

The potentials of using automatically inferred specifications in the form of invariants (pioneered by [10, 12]) are endless. For example, many established tools such as [5, 11, 16, 17, 31] construct different interpretation and views of a given system based on the automatically generated traces and invariants to build an architectural view of the system. However, reported invariants are usually noisy (many uninteresting or misleading invariants are reported) or limited in scope, which could affect the results of these tools. Thus, the accuracy of the invariant mining techniques is highly important above all other current possible limitations.

Architecturally, different people look at a system from different levels of abstraction to understand it. These levels of abstraction express specific scenarios of a system's behavior. For example, Lorenzoli et al. [21] address how different data inputs could lead to different behaviors. Generalizations of the inferred model in that case would lead to a less accurate result. Another factor is initial state objects. For example, as Krka et al.[16] shows, a stack initialized with size zero, will be behaving in a different way than any other stack of a positive size. Considering these effects during target system analysis will constructs a more thorough description of the target system. We do not attempt to address such data input factors; instead, we address other abstraction limitations in Daikon.

Accuracy limitations and noise show their significance when trying to construct transition models based on the inferred invariants [11, 16, 17] for complicated yet widely used applications. One can ask two simple questions 1) do the inferred invariants describe all objects interfaces? And 2) does the inferred specification describe the system as a whole? The answer to the first question will mostly be yes (depending on the given driving test to observe the system), while the answer to the second depends on how complicated the system is. Current mining techniques are blind to the effect of objects on each other that are written according to common coding styles or design patterns such as *Observer Pattern*, *MVC*, *State Pattern*, or *Command Pattern*. In looking closely to the essence of the problem and its common characteristics, we found that challenges of observing dynamic data structures (DDS) caused the absence of specifications that describes the system as a whole.

The nature of DDS makes them very useful for many functional and non-functional requirements. For example, in the Observer Pattern case keeping a list of observers allows for modifiability where an observer can be included at any point of time in the program execution or removed. Yet, this very nature of flexibility is what makes it difficult to observe the effect on elements of a DDS an object has. The effect could be anything like manipulating the *state* of elements in a DDS (changing a value of an element's variable) or changing the *size* of the DDS (adding or removing an element). Leading and recently published specification miner techniques are only observing a very limited set of well know Java's library DDS and they are only observed in terms of *size*. Any manipulation of the *state* of complex elements (non-primitive) in a DDS or *size* changing of an excluded DDS is unobserved. Thus, a complete view of how the system modules are interacting with each other is absent.

Based on our observation of the result of current inference techniques such as [12], we found that the issue is not the core functionality of inference process; rather it is a limitation in providing the inference engine with comprehensive traces to infer more accurate invariants. The instrumentation process for most of the inference systems lack any intelligence. This phase considered as one that should blindly provides data (traces) about the target applications regardless of the data's meaning or shape. However, it is well understood that when the instrumentation phase includes some understanding of the observed data and provides some clues to the inference engine, better specifications are likely reported. For example, Guo et al. [14] introduced a technique that groups variables based on their abstract types (e.g. `int time` is in time group and `int temp` is in temperature group, thus they should not be compared against each other) before feeding it to the inference engine. This minor change allowed the inference engine to report less noisy invariants of greater value to humans.

Current instrumentation techniques observe each object's arguments, fields, and return variables trees with depth of two. A tree fixed depth needs to be set to avoid loop references and extremely large trees. Despite it being arbitrary value, in practice the given depth (depth = 2) proved to be good for inferring thorough specification compared to others. A violation, however, of the given depth is observed on established instrumentation tools (e.g. Chicory) when encountering any complex field (e.g. Dynamic Data Structure). These tools settle for little information about the given complex field such as the number or names of elements in the DDS instead of constructing a comprehensive view of them. We think that such behavior should only be acceptable if the observed complex field (whether DDS or any other possible complex field) is encountered at the deepest level of the given object information tree, at which we only need to know general information about the complex field rather than exploring its elements or sub-fields that introduce a new level in the information tree, thus violating the depth value.

2.2 Application Example

Different design patterns introduce different complexities for instrumentation tools. It is common, for example, in the *observer* pattern for actions on objects to be triggered by a publisher that would affect multiple listeners. Such flexibility is made possible by the usage of DDS. To illustrate such behavior, we constructed an example that demonstrates a real world and widely used coding style. However, for space limitations, we present it in its simplest form [2]. Our example consists of two simple classes `Modifier` and `Receiver`.

```
public class Modifier {
  public List<Receiver> receivers = new ArrayList<Receiver>();

  public void addReciver (Receiver rcv){
    receivers.add(rcv);
  }

  public void modify (){
    for(Receiver rcv:receivers)
        rcv.increment();
  }
}
```

Example 1: Class `Modifier`

[2] Actual examples from deployed software systems follow in section 4.

The `Modifier` as shown in Example 1 allows for adding unlimited number of `Receiver`s to its DDS of type `ArrayList`. In addition, it allows for manipulating the set of receivers it has at any moment in the program lifetime by calling the method `increment()` on them through the interface `modify`.

```
public class Receiver {
  public int internalValue = 0;

  public void increment(){
    internalValue+=1;
  }
}
```

Example 2: Class `Receiver`

The `Receiver` class in Example 2 is even simpler. It has a variable `internalValue` of type `int` that is always initialized to 0. Moreover, it provides a public interface `increment()` to modify its only variable value by incrementing it by one each time the method is called.

2.3 Traditional Inference Results

Given the simple `Modifier-Receiver` example, it is easy to see how each method of each class is interacting within the system scope. Moreover, it is also easy to see the relationship between the `Modifier` and `Receiver`. That is, at any time of calling `Modifier.modify()`, any `Receiver`'s `internalValue` available in the `Modifier.receivers` array list would be incremented by 1.

```
===========================================================================
PaperExample.Modifier.addReciver(PaperExample.Receiver):::EXIT
this.receivers == orig(this.receivers)
rcv.internalValue == orig(rcv.internalValue)
size(this.receivers[])-1 == orig(size(this.receivers[]))
rcv.internalValue == 0
this.receivers[rcv.internalValue] has only one value
orig(rcv) in this.receivers[]
this.receivers.getClass().getName() ==
        orig(this.receivers.getClass().getName())
===========================================================================
PaperExample.Modifier.modify():::ENTER
this.receivers[] contains no nulls and has only one value, of length 4
this.receivers[].getClass().getName() == [PaperExample.Receiver,
        PaperExample.Receiver, PaperExample.Receiver, PaperExample.Receiver]
size(this.receivers[]) == 4
===========================================================================
PaperExample.Modifier.modify():::EXIT
this.receivers == orig(this.receivers)
this.receivers[] == orig(this.receivers[])
this.receivers[] contains no nulls and has only one value, of length 4
this.receivers[].getClass().getName() == [PaperExample.Receiver,
        PaperExample.Receiver, PaperExample.Receiver, PaperExample.Receiver]
size(this.receivers[]) == 4
this.receivers.getClass().getName() ==
        orig(this.receivers.getClass().getName())
===========================================================================
PaperExample.Receiver.increment():::EXIT
this.internalValue >= 1
this.internalValue - orig(this.internalValue) - 1 == 0
```

Example 3: Inferred invariants of the `Modifier-Receiver` example based on the Daikon's original instrumentation front-end `Chicory`.

We wrote an expressive and exhaustive test to help the inference tools (in our case Daikon [12]) infer as accurate results as possible given its original instrumentation tool (Chicory). As shown in Example 3, Daikon was able to report specifications that expressively describe the classes' behavior and their interactions as long as they are not contained within complex data structures.

For example, despite the noise, Daikon was able to infer that `Modifier.addReceiver(Receiver)` has the general effect of increasing the size of the `ArrayList` by one each time the method is existed as shown in line-23. Line-52, also, shows that the `Receiver.increment()` method has the effect of incrementing the `internalValue` by one. However, none of the inferred invariants in lines 28-40, which describes `Modifier.modify()` behavior at the observed entrance and exit points, can describe the effect of the given method. In fact, those invariants suggest that the `Modifier.modify()` method is side-effect free both on its class scope and the system scope. This misleading inference is caused by not looking at the internal fields of the objects stored in the `Modifier.receiver` array list.

3 INSTRUMENTATION WITH DDS IN MIND

In this section, we demonstrate the challenge of tracking DDS in its abstract form highlighting the obstacles and the way we overcome them. Section 3.1 presents the challenges and the possible benefit we would gain by overcoming those challenges. In the following two sections (3.2 and 3.3) we describe in detail how are we observing the elements of possible DDS and the rationale behind the once we report. The last section 3.4 discusses the possible DDS that we could apply on our technique.

3.1 Challenges and Benefits

As we stated in the introduction, the very dynamic nature of the well-known and widely used dynamic data structures is what makes it useful to the programmer and at the same time challenging to observe with inference tools. The usefulness of those kinds of data structures is clear. However, to understand the challenges we need to understand how current mining techniques are observing the target application. Daikon [12], for example, expects a well-defined structure of variables to track that defines a given method at an entrance or exit point.

The variables tree structure in Figure 2 shows the actual tree structure Chicory (current instrumentation tool for Daikon) will generate for the method `Modifier.addReceiver(Receiver)` from our last code example at an exit point. Also, the figure shows a potential possible branching represented in gray just to fully explore and understand the technique (will elaborate on this in the following paragraph). The actual nodes, as shown is the diagram, are constructed based on the passed arguments and object fields (another branch would be constructed if the method `addReceiver` has a return value). User defined objects are the only fields or arguments that are further explored. The variables `receivers` and `internalValue` are a DDS and a primitive respectively, thus no further exploration conducted. The exploration process in the case of the user-defined object thus halts and no more variables to track added.

As the gray potential example shows, we are assuming that the `Modifier` has a user-defined object *a*. The object *a* has another user defined object *b* and *b* has an instance of *c*. Even though *c* also has an instance of *a* the exploration process stops at *c* to avoid such infinite loop of references. At such case, it is clear why an arbitrary fixed depth is necessary. Even though the depth 2 could result on the loss of valuable insight about the system, it is a widely used

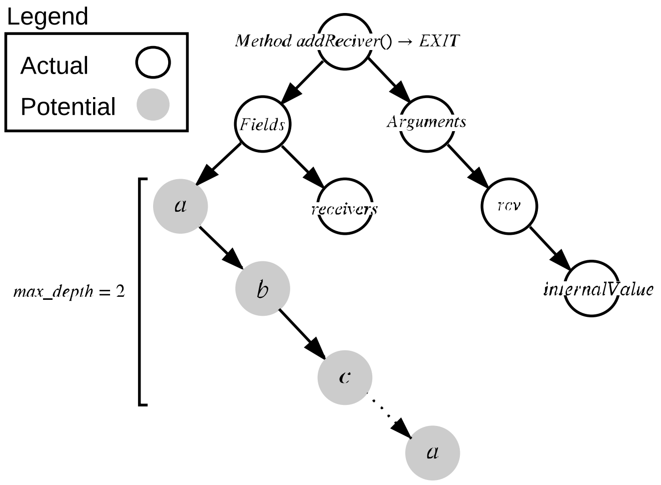

Figure 2: Illustrating the *actual* variables tree structure of method `addReceiver` at an exit point using Chicory. Also, because the object does not have user-defined object, we show a *potential* construction of the variables tree by Chicory assuming the object has a user-defined field *a*.

value and arguably helps exploring a target application thoroughly. Thus, we can see that when Chicory encounters complex fields (e.g. DDS) it violates the given depth rule by discarding the exploration process of that field.

In Chicory, the given structure is defined as soon as a method is invoked and must never change. All future invocations of the given method will use the given tree structure of the method to track *traces* (values of variables). However, this established behavior is ignoring the fact that the `receivers` DDS could hold some interesting element to track at a future point in the program execution. Also, it ignores the fact that an element could exist at a point in time in a DDS and then be removed before the application halts. These behaviors makes it challenging to track DDS's elements.

Time in the instrumentation stage represented on the sequence in which a method is invoked. For example, consider the method `addReceiver`, we can represent the invocation sequence as $\{addRecevier_1, addRecevier_2, ..., addRecevier_n\}$, where $addRecevier_1$ is the first time the method was invoked and $addRecevier_n$ is the last time the method was invoked. In our application it is possible to have the DDS $receivers = \{e_1, e_2, ..., e_m\}$ at time $addRecevier_1$ and the same DDS $receivers = \{e_1, e_2, ..., e_{m+1}\}$ at $addRecevier_n$. Hence, the given technique to observe variables structure is limited.

Chicory records a dynamic sequence of values assigned to variables, but the system fields structure (the set of variables it tracks) is static. Dynamically tracking the values only, could be beneficial in observing behaviors like *size* change (e.g. observing that method `addReceiver` increases the size of the given array list). However, in addition to violating the given *max_depth* value, Chicory drives Daikon to miss the opportunity of observing the *states* of the elements in a DDS (e.g. observing that the method `modify` is incrementing each `internalValue` of the existing `receivers` by one). A fully dynamic instrumentation tool is required to show such effect.

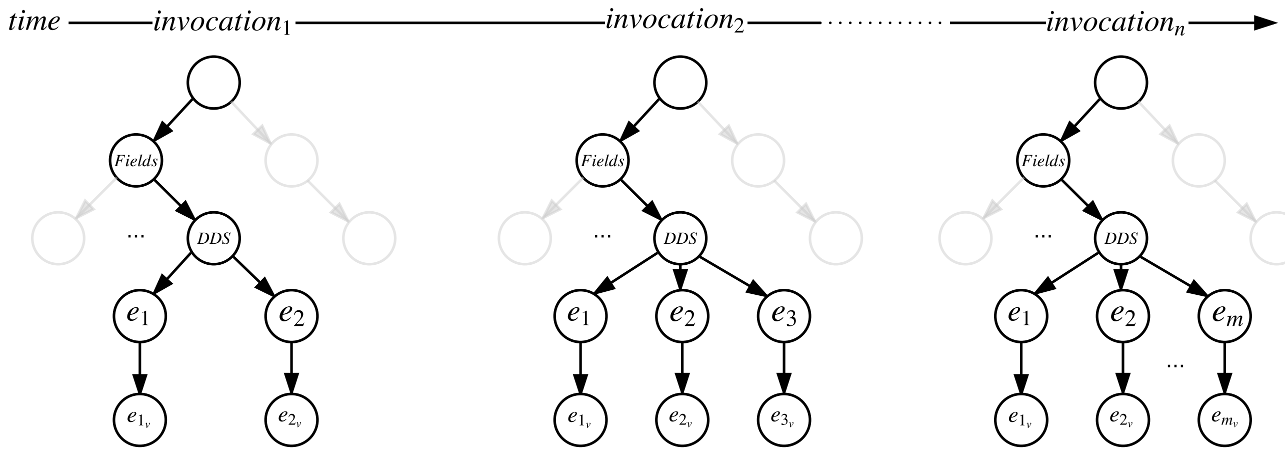

Figure 3: A sequence of possible variable trees of a method at entrance or exit point.

3.2 Tracking System Flow

To overcome the challenge of the changes in variables structure of a given method execution point, not only the values of the variables should be evaluated each time a method is invoked, but also the variables structure should be regularly evaluated. For example, each time the method `addReceiver` is invoked a new variables tree structure must be constructed and buffered. Each time a method variable tree is constructed, we observer possible current elements of a given DDS treating them as user-defined classes. For each buffered method variable tree, we record traces (values).

The invocation sequence shown in Figure 3 illustrate the steps of constructing variable trees of a method at each possible invocation. Given any method at its entrance or exit point, in addition to the observation of user-defined fields, we observe all the elements of a possible DDS. As long as we are within the given *max_depth*, we explore the elements' fields if they are non-primitives. Even if the elements themselves are a type of DDS, we still observe them by only exposing their elements (since at that point we are surely exceeding the given depth).

3.3 Unifying the Knowledge about the System

The presented behavior in section 3.2 should ensure that we can always observe elements of a possible DDS. Moreover, it ensures that we keep track of the elements' fields values similar to any object that is associated with the current method. However, such design introduces issues like the inconsistency in the variable trees. Thus, it is not possible to feed it to an inference tool designed for fixed variable structures. To mitigate this, we introduce a unifying stage to unify all the variable trees of a method considering elements additions and removal from the DDS.

The essence of the issue is that it is possible to have (1) an element be introduced that was not present at an earlier invocation or (2) an element that exists before in the DDS but was removed at a later point on time. For the first case, we found out that we could treat a newly introduced but never later removed element as a variable that is not instantiated yet. For example, from Figure 3 we can safely unify the variables tree from $invocation_1$ with the variables tree from $invocation_2$ by considering the tree from the latter invocation for both invocations. And updating the traces generated for $invocation_1$ by introducing a nonsensical trace for the elements e_3 and e_{3_v}. For these two variables, we can consider the values at $invocation_1$ as nonsensical, which is how Chicory handles uninitialized variables. However, for the second case where an element is present in an early variable tree then be removed in a later one. Because Daikon was not designed to track an evolving set of fields, a variable cannot be removed or given a nonsensical value once it is initialized. We believe that in such case the element's *state* is not as interesting as the fact that it was taken out (*size* change). Thus, we take the variable out from all trees as well as its traces for that particular method in which its existence is not always guaranteed. Therefore, we only observe the *size* of the DDS but not the *state*.

This behavior should guarantee that all unified variable trees and traces are readable by an inference tool. Moreover, the technique should guarantee that the behavior of the target application is intact (no values are wrongly changed or introduced).

3.4 Analysis of Applicable DDS

In building an instrumentation tool, a good effort needs to be focused on the language syntax and semantics to dynamically observe any target application. Another aspect that needs to be focused on is the commonly used libraries for the targeted language. For Java, the Java Development Kit is the standard library that provides many generic implementation of DDS. Chicory is implemented in a way that partially takes advantage of the already known implementation of standard DDS. For example, when encountering a usage of the `List` implementation of the JDK, Chicory is only designed to look at how many elements are in the list and what are they. It does not attempt to further analyze the elements of the list.

Embracing the prior knowledge we know about these libraries to allow our instrumentation tools to provide better traces should be the norm. Especially, that current instrumentation tools are designed to take advantage of these libraries interfaces to derive some knowledge about DDS (like the Chicory case with lists). It is possible to know the generic type of the elements in the DDS using the target application and reflection from Java. Thus, using more known interface to get, analyze and construct structure variable trees of the DDS elements is feasible.

There are different types of DDSs in the JDK. We analyze all the concrete DDSs types that extends or implements the `Collection` and `Map` interfaces. Given that we can identify each element in the given DDS by recording its `hashCode` (the possibility of collision is minimal), then it is safe if the order of elements is changed from one observation time to another. Also, given that the given interface insures we would be able to get the elements as an array guarantees that all implementations provide a known method for element retrieval. Thus, we believe that most of the concrete classes that implements the `List` interface (including `Vector<T>`, `Stack<T>`, `ArrayList<T>`, `LinkedList<T>` and `CopyOnWriteArrayList<T>`), the `Set` interface (including `HashSet<T>`, `TreeSet<T>`, `LinkedHashSet<T>`, `CopyOnWriteArraySet<T>` and `ConcurrentSkipListSet<T>`), the `Queue` interface (including `ArrayDeque<T>`, `PriorityQueue<T>`, `ConcurrentLinkedQueue<T>`, `ArrayBlockingQueue<T>`, `PriorityBlockingQueue<T>`, `LinkedBlockingDeque<T>`, `LinkedBlockingDeque<T>` and `LinkedTransferQueue<T>`), and the `Map` interface (including `Hashtable<K, V>`, `HashMap<K, V>`, `TreeMap<K, V>`, `WeakHashMap<K, V>`, `LinkedHashMap<K, V>` and `IdentityHashMap<K, V>`) are applicable cases. We only exclude implementations that allow of concurrent access of the DDS's elements. For our prototype, we only implement the observation mechanism of the `List` interface as a proof of concept.

4 EVALUATION

This section presents the evaluation methods and results of applying our approach for invariants mining. We discuss the reasoning behind the inapplicability of the precision and recall measurement for our evaluation in Section 4.1 and describe the alternative method for evaluating our approach in Section 4.2. Section 4.3 shows the candidate target applications and the criteria for selecting them as well the results from our evaluation.

4.1 Precision & Recall

In research literature on dynamic invariant mining, precision and recall has being the main approach for evaluating accuracy of existing techniques. In a nutshell, to evaluate precision and recall for any target application humans would need to develop a set of specifications manually and consider that as a ground truth. Then, given the results from the introduced inference technique the *precision*: how many of the reported invariant are relevant compared to the given ground truth and *recall*: how many relevant invariants are reported given the ground truth are measured.

Precision and recall common usage on evaluation made us consider already established evaluation techniques [20, 27]. However, these proposed techniques are either require human intervention [20, 27], which we are trying to avoid, or focus on different quality dimensions such as scalability and robustness [20].

Such evaluation approach suffers from multiple issues. First, even if different people revise the established ground truth, it is still open for debate and can be biased for whatever perspective the inference system is focusing on. Second, it involves a lot of human effort when applied on large real world applications; in practice, this limits evaluation of precision and recall to small examples. Third, we prefer a *ground truth* that is not dependent on developer's opinion, which could differ from developer to developer. An approach that can be easily scaled and fully automatic to be used against real world applications is necessary.

4.2 Purity with Dynamic Analysis

Although we do not have access to complete ground truth, we can evaluate the extent to which invariant detection agrees with partial ground truth that can be extended automatically by other means. The notion of *purity* is already established and well defined in static analysis domain. Finding *pure* methods has never being the goal of dynamic analysis techniques. Rather, they report invariants that highlights what is effected by a method in terms of fields or passed arguments. However, the essence of the problem we observed is the lack of reported effect on non-*pure* methods. Thus, we believe that it is fair to say if a given method is not *pure* and the given invariant miner approach fails to capture any effect, then the invariant miner is inaccurate. Such evaluation can be conducted automatically and eliminate any biased.

In looking for a tool that would establish the ground truth using static analysis, we found two candidates [26, 28]. Both applications are research-based tools. Thus, in our decision making for which one to use, we focused mainly on which one is more likely to work against all target applications. Given that Jpp [28] has not being update since 2006 and that it failed against minor examples, we decided to go with the jPure [26] even though it is too has not being updated lately (last update was in 2011). In our evaluation, we observe each target class using jPure. Given the results from eChicory and the results from Chicory, we say the technique that is closer in agreement to the number of pure method to the one from jPure is more accurate.

4.3 Experiment

4.3.1 *Artifact Selection Criteria.*

In selecting target applications, we selected GitHub as the source of artifacts. We considered Java applications of the size between 2K and 10K LOC. The given size increases the probability of encountering interesting implementation that adapts common design patters. Moreover, it ensures that we are targeting modest but non-trivial applications. Evaluation of dynamic invariant detection with large software systems is rare to nonexistent in the research literature. Ernst et al. [12] in the original Daikon paper evaluate their approach against an application of size no larger than 563 LOC. Krka et al. [16], evaluate their approach against examples of smaller size than the original `StackAr` example that is widely used to evaluate specification miners.

In addition to the size of the target application, we depended on the issues tracker, pull requests, and wiki of the given repository looking for insight that the given artifact implements one of the design patterns we are targeting. This technique does not guarantee

Application	Description	Class	# of Methods	Targeted Design Pattern
Mockito [4]	Mocking framework for unit tests in Java	InvocationNotifierHandler	7	Observer Pattern
Apache Struts [1]	Framework for creating Java web applications	DefaultActionInvocation	29	MVC
		DefaultUnknownHandlerManager	7	
		ConfigurationManager	16	
		VelocityManager	18	
		SimpleTextNode	17	
		SimpleAdapterDocument	43	
JabRef [3]	BibTeX Management application	EntryEditor	22	MVC
		CleanupActionsListModel	8	
		UndoableModifySubtree	4	
		ImportInspectionDialog	21	
Zeppelin [2]	A web based interactive data analytic tool	Folder	23	Observer Pattern
		Notebook	45	
		NotebookRepoSync	31	

Table 1: Selected target applications, their classes and the number of observed methods on them.

that we can find a useful implementation or it actually reflects the available source code. However, we mitigate that by looking at artifacts with more stars (GitHub indicator of popularity) or maintained by a well-known organizations (e.g. Apache Software Foundation). We also targeted applications with high test coverage (if reported) to avoid test incompleteness.

The given search criteria resulted in four different artifacts thus far (see Table 1). When observing a given application we do not attempt to run eChicory (or Chicory) against all available classes in the target application. Instead, we look for classes that would express our contribution and test them. Our technique should result in similar results as in original Chicory for all other classes. Any change could be in reporting invariants that are *true* but not interesting (e.g., a variable in a DDS has not changed after invoking a method).

4.3.2 *Artifacts Analysis.*

To evaluate our approach we ran eChicory and Chicory against each class from Table 1 alone. Ideally, each class should have its own designated unit test. However, in practice this is often not the case. Thus, in the case where a class has no designated unit tests we ran all available unit tests within a module or the whole application to increase the probability of testing the given class thoroughly. We then feed the given traces from eChicory and Chicory to the same version of Daikon (released January 5, 2017). From the resulting invariants we look at whether a method has any effect of any sort (e.g. a variable value is changed) for each given invariant. A method that has invariants shows an effect is *non-pure*. Otherwise, the given method appears *pure* based on the given instrumentation technique.

We conducted our experiment on a Linux virtual machine that has 5GB of memory and uses the Intel i7-3667U CPU on the host machine. In the only example from `Mockito` there exists seven methods. A constructor, which clearly sets the only two fields of the class, two expectedly *pure* getters, a setter for one of the fields, and three special purposes methods (`handle`, `notifyMethodCall`, and `notifyMethodCallException`). Each one of the three special purpose methods are iterating over the DDS elements calling methods that eventually changes the elements' *status*. Based on *jPure*, the class has no *pure* methods at all. Both instrumentation tools detect that the constructor is not *pure*. Moreover, both agree on showing that the getters are *pure*. However, neither Chicory nor eChicory is able to show the effect of the only setter and thus consider it *pure*. Most importantly, to the contrary of Chicory, eChicory was able to help Daikon infer the effect of the three special purpose methods. Thus, all of the three methods are non-*pure* according to eChicory, which is closer to the result report by *jPure*.

For the other applications, our testing was not fruitful for many different reasons. Inadequate unit tests of these applications is the general issue. In `Apache Struts` for example, the classes `DefaultUnknownHandlerManager`, `VelocityManager`, `SimpleTextNode` and `SimpleAdapterDocument` are all either not hit at all as a class when we ran all the available unit tests or partially hit (small portion of the available methods). For `JabRef` the issue is even worse where all the unit tests related to the package we are interested in (the GUI package) are broken. The application's development team confirmed that the GUI related tests are pixel sensitive and currently unmaintained. `Zeppelin` also has inadequate tests for the class `NotebookRepoSync` where two of the interesting methods are missed, while the third one is observed but in a naive inexpressive way.

Another less common issue is the possibility of having trivial implementations of the DDS element's type or even using a mock representation of the elements type when testing a class. In such case the methods of the classes were sufficiently hit, but the elements of DDS do not have any interesting fields as they would when used in production. Sometimes the interesting behavior was hidden because of the usage of a mock representation of the DDS elements. For example, the class `ConfigurationManager` from `Apache Struts` has a DDS of type `ContainerProvider`, which is an interface. Two different classes implement the given type. In both implementations, the method `destroy()`, which the subject class `ConfigurationManager` calls, is not implemented. Thus, in any scenario in which the class `ConfigurationManager`

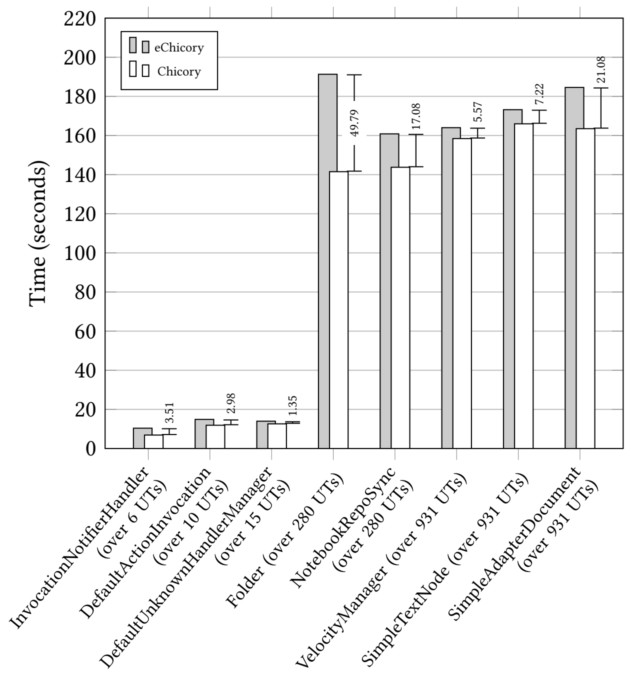

Figure 4: Traces collection performance difference between `eChicory` and `Chicory` over each class and the number of unit tests (UTs) used to observe it.

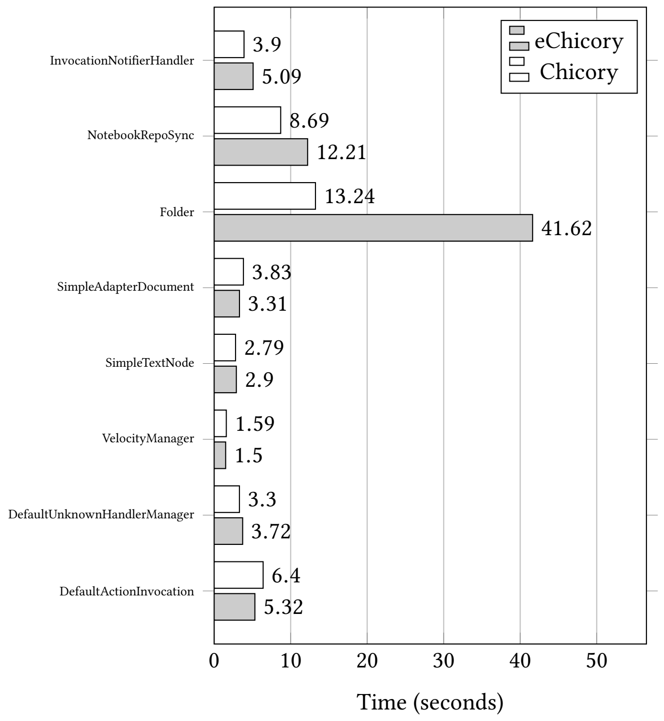

Figure 5: Time taken by Daikon to read and generates invariants from traces provided by `eChicory` and `Chicory`

would be tested the traces will not result in any interesting insight. Similar observation is applicable on the `Folder` class from `Zeppelin`.

A different issue could be the complete absence of elements in the DDS when a class is tested. For example, in the `DefaultActionInvocation` class from `Apache Struts` case, the test did not attempt to add any elements to the field `preResultListeners`. Such testing behavior would hit the given class but would not demonstrate our approach, making the resulting trace from eChicory and Chicory identical.

Finally, in a rare case our prototype fails against some classes. For example, when running our tool against the class `Notebook` from `Zeppelin` traces were generated successfully. However, when feeding those traces to Daikon, an exception is thrown by Daikon with no clear message. We fixed many of these types of problems and rarely see them lately, but the `Notebook` case is one of those that are still not solved.

4.3.3 *Performance Insight.*

We believe that accuracy of dynamic software analysis techniques is the major issue in the field. Performance in Daikon [12] is influenced by many factors (e.g. number of variables in scope, number of times a method is observed, and number of defined program points (method exist or entrance)). Improving the performance would usually mean a trade off on accuracy. However, even if a technique is performing very well, if the reported invariants are not useful because of inaccuracy issues, then it would not be relevant whether a technique is performing well or not. Nevertheless, we wanted to give a brief insight into how our approach is affecting the performance of the system analysis process. We show this from two different perspectives, 1) the effect of our approach on the trace collection process 2) the effect of collected traces on the invariants inference engine.

The trace collection on eChicory is broken into two different stages (trace collection and trace unification) as explained before. We combine the time taken by these two different processing stages to show the performance of eChicory. In Figure 4, we show the wall clock time taken by eChicory and Chicory when observing each class of each candidate application. We only omit `Struts.ConfigurationManager` and `Zeppelin.Notebook` since the first one is not worth testing given that methods are not implemented and we fail for the second one. From the chart, it is clear that the performance of eChicory is generally acceptable. The only significant difference can be seen when testing the classes `Folder` and `SimpleAdapterDocument` from `Zeppelin` and `Apache Struts` respectively. The difference in performance for the `Folder` class is less than 50 seconds, while it is slightly over 21 seconds for the `SimpleAdapterDocument` class. The large number of observed elements in the DDS causes this observable difference. Thus, we can say that the performance of our approach directly influenced by how many elements in a given DDS and how many fields are there for each element.

The effect of our approach on the invariant inference process could be seen in Figure 5. From the figure, the effect on performance is negligible except for the `Folder` class traces. The significant different that is shown when observing the `Folder` class is explained by the same reason as performance deficit when generating the traces. The more elements in DDS we get during testing the more time it would be required to generate traces or use them for invariants inference.

The shown performance records are not fully representative yet of our approach. This is because the majority of the given classes eventually considered as limited. For example, when testing the class `VelocityManager`, there were no traces observed by either instrumentation tools. Similarly, for classes `SimpleTextNode` and `SimpleAdapterDocument` the given unit test were not useful as they did not access any DDS elements. The only class that could give us an acceptable indication of performance is the `InvocationNotifierHandler` class. Yet, even this class is trivial; we are working on observing additional test subjects to have better evaluation of our contribution.

5 THREATS TO VALIDITY

In this paper, we discussed the benefits of observing DDS when dynamically analyzing a system for more comprehensive and accurate results. It is important to highlight that we do not attempt to discover DDS implemented by users. We only take advantage of the prior knowledge we know about well-known and widely used libraries implementations of different types of DDS to adopt an instrumentation tool to it. Any new type of data structure would need to be thoroughly analyzed to see if it is applicable for observation. Only then, we would need to enhance the instrumentation tool to be able to identify the new type of DDS, check its properties and assimilate its elements. Observing any unknown implementation of DDS models is beyond or goals and is covered in Section 6 in more depth.

Another threat to the validity of our contribution is the introduction of more noisy specifications. In dynamic inference systems, reported invariants that are true but not useful are noisy invariants. For example, each time the method `addReceiver` from our earlier example is called, the size of the DDS `receivers` increases by one. However, the value of all the other variables (e.g. `receivers[0].internalValue` of each already existing receivers) are not changed. Thus, the inference system will report for each variable that when the method is exited, the variables are not changed. This is a *true* invariant but is considered noise since it could be inferred easily by developers. An effective solution should be applied to the inference engine rather than the instrumentation tools. Noise does not only arise from variables that are seen in DDS. Rather it could be introduced by any simple field of a given object. Variables found on DDS elements only make such issue stand out because of the increased number of tracked variables.

If a solution would be introduced at the instrumentation stage, it would by analyzing the behavior of variables between a method entrance and exit, then make a decision about reporting it to the inference engine. For example, if a variable's value never changes between the method entrance and exist throughout the whole given traces it could be removed. However, this could interfere with the object level invariants and mislead the inference engine. We decided to keep the instrumentation phase of the inference process unchanged and consider this an opportunity for future work.

6 RELATED WORK

Mining specification and dealing with dynamic data structure have being explored by many different people. However, considering both at the same time is not heavily explored.

In their work, White et al. [30] established a very interesting technique to identify dynamic data structures in C. Their work is focused on observing user-defined objects that implement an already established and well-defined DDS. Even though the value of this work is clear, they do not attempt to report an understanding of the system given those defined dynamic data structure. We believe that if our work is merged with their, in that we can define user implemented dynamic data structure while at the same time explore how these affect other paths of the system, then a very generic solution would exist. A generic solution would means we are not limited by the well-defined DDSs in the Java standard library or any other library for any other programming language.

7 FUTURE WORK

We believe there is much room for enhancement in the field of dynamic program analysis in general. However, in relation to the work we presented, we would like to tackle two important problems in the future.

Noise is an issue we need to address. Whether the reported invariants are based on Chicory or eChicory, the existence of noise makes invariants hard to follow for people whom are looking to understand the system (have no prior knowledge about it). Introducing a solution at the instrumentation stage can negatively affect the accuracy of the inference engine results as discussed in section 5. Thus, applying a solution to avoid reporting noisy invariants on the inference engine is one issue we would like to tackle in the future.

Second, current instrumentation techniques are not only blind to common coding styles or design patterns because of not observing DDS. There are different design patterns such as *Command* and *State* patterns that does not involve DDS, yet instrumentation tools fail to provide comprehensive trace to allow inference engines report accurate results. One possible cause of limitation is the interchangeable ability of a field in an object. For example, a field in a target class can be of an interface implemented by multiple concrete classes. The given field can be pointing at any of the concrete classes at any given time. Thus, it is difficult to know which variables of which concrete class to track. In such case, current specification miners violate the given depth to observe variables in favor of determinism. We aim at further analyzing and applying a solution for each of the given design patterns in the instrumentation side to allow for more accurate and system level reported invariants.

8 CONCLUSION

In this paper, we highlighted the issue of out-of-scope effect that current mining techniques suffer from. These are not corner cases. Rather these are cases show up by using well-known and widely used design patterns. In a deeper look into the problem, we identified that lack of good methodologies of observing already defined DDS is one cause of the accuracy issue. We proposed a methodology to expand the scope of exploration and the number of properties to track on a system dynamically instead of statically track variables.

We implemented a prototype of a solution for observing DDS as proof of concept each time a method in the target application is invoked. Moreover, we introduced a solution to unify collected variable trees to make it possible to use the generated traces with existing inference engine.

Finally, we showed by example that our solution finds useful invariants missed by current techniques. Current instrumentation techniques are blind to the effect of some very basic coding styles. We also attempted to test our solution against more applications, which led us to reach the conclusion that the testing state in the open source community is not ideal. Further, we provided an insight about the performance of our approach against current state of the art techniques.

REFERENCES

[1] 2017. Apache Struts. (2017). https://struts.apache.org/

[2] 2017. Apache Zeppelin. (2017). https://zeppelin.apache.org/

[3] 2017. JabRef. (2017). https://www.jabref.org/

[4] 2017. Mocking Framework for Unit Tests in Java. (2017). http://site.mockito.org/

[5] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 177–188. DOI: http://dx.doi.org/10.1145/2931037.2931049

[6] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2013. Unifying FSM-inference Algorithms Through Declarative Specification. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 252–261. http://dl.acm.org/citation.cfm?id=2486788.2486822

[7] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, Szeged, Hungary, 267–277. DOI: http://dx.doi.org/10.1145/2025113.2025151

[8] Christoph Csallner and Yannis Smaragdakis. 2006. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 245–254. DOI: http://dx.doi.org/10.1145/1146238.1146267

[9] Christoph Csallner and Yannis Smaragdakis. 2006. Dynamically Discovering Likely Interface Invariants. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 861–864. DOI: http://dx.doi.org/10.1145/1134285.1134435

[10] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, Leipzig, Germany, 281–290. DOI: http://dx.doi.org/10.1145/1368088.1368127

[11] Guido de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. 2012. Automated Abstractions for Contract Validation. *Software Engineering, IEEE Transactions on* 38, 1 (Jan 2012), 141–162. DOI: http://dx.doi.org/10.1109/TSE.2010.98

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, Los Angeles, California, USA, 213–224. DOI: http://dx.doi.org/10.1145/302405.302467

[13] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1-3 (Dec. 2007), 35–45. DOI: http://dx.doi.org/10.1016/j.scico.2007.01.015

[14] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. 2006. Dynamic Inference of Abstract Types. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 255–265. DOI: http://dx.doi.org/10.1145/1146238.1146268

[15] Sudheendra Hangal and Monica S. Lam. 2002. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 291–301. DOI: http://dx.doi.org/10.1145/581339.581377

[16] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic Mining of Specifications from Invocation Traces and Method Invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, Hong Kong, China, 178–189. DOI: http://dx.doi.org/10.1145/2635868.2635890

[17] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. 2010. Using Dynamic Execution Traces and Program Invariants to Enhance Behavioral Model Inference. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 179–182. DOI: http://dx.doi.org/10.1145/1810295.1810324

[18] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, and Michal Young. 2013. Second-order Constraints in Dynamic Invariant Inference. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 103–113. DOI: http://dx.doi.org/10.1145/2491411.2491457

[19] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 11th Annual Information Security Symposium (CERIAS '10)*. CERIAS - Purdue University, West Lafayette, IN, Article 5, 1 pages. http://dl.acm.org/citation.cfm?id=2788959.2788964

[20] D. Lo and S. c. Khoo. 2006. QUARK: Empirical Assessment of Automaton-based Specification Miners. In *2006 13th Working Conference on Reverse Engineering*. 51–60. DOI: http://dx.doi.org/10.1109/WCRE.2006.47

[21] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic Generation of Software Behavioral Models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 501–510. DOI: http://dx.doi.org/10.1145/1368088.1368157

[22] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. *SIGPLAN Not.* 40, 6 (June 2005), 48–61. DOI: http://dx.doi.org/10.1145/1064978.1065018

[23] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. DOI: http://dx.doi.org/10.1145/1273442.1250746

[24] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2016. Learning API Usages from Bytecode: A Statistical Approach. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 416–427. DOI: http://dx.doi.org/10.1145/2884781.2884873

[25] Robert O'Callahan, Robert O'Callahan, and Daniel" Jackson. 1997. Lackwit: A Program Understanding Tool Based on Type Inference. *IN PROCEEDINGS OF THE 19TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING* (1997), 338–348. DOI: http://dx.doi.org/10.1.1.36.9375

[26] David J. Pearce. 2011. *JPure: A Modular Purity System for Java.* Springer Berlin Heidelberg, Berlin, Heidelberg, 104–123. DOI: http://dx.doi.org/10.1007/978-3-642-19861-8_7

[27] M. Pradel, P. Bichsel, and T.R. Gross. 2010. A framework for the evaluation of specification miners based on finite state machines. In *Software Maintenance (ICSM), 2010 IEEE International Conference on.* 1–10. DOI: http://dx.doi.org/10.1109/ICSM.2010.5609576

[28] Alexandru Sălcianu and Martin Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. Springer-Verlag, Berlin, Heidelberg, 199–215. DOI: http://dx.doi.org/10.1007/978-3-540-30579-8_14

[29] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring Better Contracts. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 191–200. DOI: http://dx.doi.org/10.1145/1985793.1985820

[30] David H. White, Thomas Rupprecht, and Gerald Lüttgen. 2016. DSI: An Evidence-based Approach to Identify Dynamic Data Structures in C Programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 259–269. DOI: http://dx.doi.org/10.1145/2931037.2931071

[31] T. Ziadi, M.A.A. Da Silva, L.M. Hillah, and M. Ziane. 2011. A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on.* 107–116. DOI: http://dx.doi.org/10.1109/ICECCS.2011.18