Extending Existing Inference Tools to Mine Dynamic APIs

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

Michal Young
Computer Science Dept.
University of Oregon
Eugene, OR
michal@cs.uoregon.edu

ABSTRACT

APIs often feature dynamic relations between client and service provider, such as registering for notifications or establishing a connection to a service. Dynamic specification mining techniques attempt to fill gaps in missing or decaying documentation, but current miners are blind to relations established dynamically. Because they cannot recover properties involving these dynamic structures, they may produce incomplete or misleading specifications. 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 that establish new relations between client and service provider. 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 evaluated our tool by applying it to selected modules of widely used software systems published on GitHub.

CCS CONCEPTS

• **Software and its engineering → Software system structures**; **Documentation**; **Software evolution**;

KEYWORDS

dynamic analysis, specification mining, design patterns

1 INTRODUCTION

Software API documentation is often incomplete, and misunderstandings of behavior can be disastrous [9]. Dynamic specification miners that extract information about observed behavior based on running code could be helpful, but only if the conjectures[1] they observe are raised to the appropriate abstraction level and include dynamically created relationships, such as registering an event listener.

Specification miners and invariant detectors extract potentially useful conjectures about program behavior from program source code [14, 15], dynamic monitoring of behavior [5, 7, 8, 10–13, 17], or both [6]. 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. 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. They are limited in detecting dynamically established relations, such as dynamic attachment of client code to libraries.

Dynamic invariant detectors can be extended to capture properties of interactions between objects whose relations are created during execution, 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 monitoring 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 Mockito [4] 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. Current invariant detectors monitor the relation between values of two variables only if they have a static relationship. Therefore, invariants reported by current invariants detectors (e.g. Daikon [8]) might lead one to believe the method has no effect, regardless of the completeness of the test suite.

We have constructed a proof-of-principle prototype instrumentation tool called eChicory (Enhanced Chicory) for inferring invariants about objects in dynamic relationships by extending the front end of Daikon (Chicory) dynamic invariant detector for Java [8]. eChicory can recognize dynamic relationships between classes and objects that arise in common design patterns such as the *observer* and *model-view-controller* patterns. While our prototype implementation is limited to Java libraries and is based on Daikon [8], the basic approach should be applicable to other programming languages and invariant detection tools.

[1]The mined specifications are called "conjectures" or "likely-invariants" because they depend on observed executions, not all possible behaviors.

WAPI'18, June 2–4, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5754-8/18/06…$15.00
https://doi.org/10.1145/3194793.3194797

2 DYNAMIC INSTRUMENTATION

Flexible APIs often support dynamic association of clients with services. Dynamic data structures (DDS) associating client code with service code provide flexibility and utility, but they present a challenge to dynamic invariant detectors. To understand why, we must consider how these systems work. Daikon [8], for example, expects a stable structure of variables to track at method entrance and exit points.

Section 4 describes evaluation of a prototype tool with deployed libraries taken from Github. In this section we illustrate the approach with a simplified example that follows a widely used pattern but consists of just two classes, `Modifier` and `Receiver` (Example 1). The `Modifier` allows adding an unlimited number of `Receivers` in an `ArrayList`. The `Receiver` is even simpler: its variable `internalValue` is initialized to 0 but can be modified through its public interface `increment()`.

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

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

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

public class Receiver {
  public int internalValue = 0;

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

Example 1: Classes `Modifier` and `Receiver`

The variables tree structure in Figure 1 shows the *actual* tree structure Chicory (the current instrumentation tool for Daikon) will generate for the method `Modifier.addReceiver(Receiver)` at an exit point from Example 1 as well as a *potential* branching to fully understand the technique. The *actual* nodes, as shown in 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 variables further explored. The variables `receivers` and `internalValue` are a DDS and a primitive respectively, thus no further exploration is conducted. The exploration process in the *actual* case thus halts and no more variables to track are added.

We are assuming that the `Modifier`, as the *potential* nodes shows, has a user-defined object *a*. The object *a* has another user defined object *b* and *b* has an instance of *c* that points to *a* making a loop of references. The unbounded referencing loop makes it 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 value and arguably helps exploring a target application thoroughly. Thus, we can see that when Chicory encounters complex fields (e.g. the DDS `receivers`) it violates the given *max_depth* rule by discarding the exploration process of that field in favor of determinability.

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). Thus, it is dynamic on values assigned to variables, but static on system fields structure (the set of variables it tracks). This established behavior ignores the fact that the `receivers` DDS could hold some interesting element to track at a future point in the program execution. Dynamically tracking only the values could be beneficial in observing behaviors like *size* change (e.g. observing that method `addReceiver` increases the size of the given array list), but it drives inference systems to miss opportunities of observing the *states* of the elements within a DDS. A fully dynamic instrumentation tool is required to show such effect.

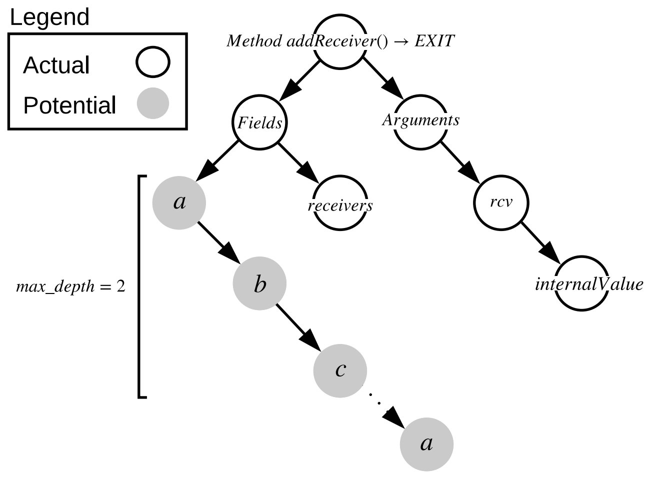

Figure 1: Chicory (Daikon instrumenter) tree of variables for method `addReceiver` labeled as *actual*, plus *potential* references not included in this simple example.

3 TRACKING FLOW AND UNIFYING KNOWLEDGE

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 observe possible current elements of a given DDS treating them as user-defined classes. For each buffered method variable tree, we record traces (values).

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 (in the case of *max_depth* = 2).

The presented new behavior 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, this design introduces issues like the inconsistency in the variables trees that cannot be used on existing inference tools. To mitigate this, we

Application	Targeted Design Pattern	Description	Class	# of Methods Tested	Number of Pure Methods		
					jPure	Chicory	eChicory
Mockito [4]	Observer Pattern	Mocking framework for unit tests in Java	InvocationNotifierHandler	7	0	5	2
Apache Struts [1]	MVC	Framework for creating Java web applications	DefaultActionInvocation	29	-	ᴀ	ᴀ
			DefaultUnknownHandlerManager	3	ᴇ	2	0
			ConfigurationManager	16	-	ɱ	ɱ
			VelocityManager	18	-	ɪ	ɪ
			SimpleTextNode	17	-	ɪ	ɪ
			SimpleAdapterDocument	43	-	ɪ	ɪ

Table 1: Selected target applications, their classes, the number of observed methods on them and the result from Chicory and eChicory. ɪ: Inadequate unit tests. ɱ: Usage of mocks to represent DDS elements. ᴀ: Absence of DDS manipulation. ᴇ: Failure on observing the class.

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 that we could treat a newly introduced but never later removed element as an uninstantiated variable. Moreover, we can consider the value of any introduced variable from the merge process as nonsensical, which is how Chicory handles uninitialized variables. However, 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. In that 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 guaranteed over all invocations Therefore, we only observe the *size* of the DDS but not the *state* in this particular case.

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).

4 EVALUATION

4.1 Evaluation Methods

We wished to evaluate the extent to which our enhancement to dynamic invariant detection was useful with widely used code bases of modest size (2k to 10k LOC) which were not constructed by us and with which we had no special knowledge or familiarity. The natural criteria for measuring the effectiveness of invariant detection are recall and precision, but we faced a problem in applying these to systems that are both unfamiliar and of significant size. They must be measured with respect to some ground truth, which in this case would be specifications. We cannot know the "real" specifications for our target applications, and manually extracting a complete set of specifications does not scale to applications of even modest size. It was imperative to find an alternative that could at least be partially mechanized.

Our goal is to characterize invariant relations maintained by updating fields in a dynamic data structure (showing effect). Some invariants can highlight a variable state in comparison to another variable (e.g. `a > b`), others can define the state of the variable (e.g. `a is null`) and few invariants can show the effect on a variable (e.g. `a = orig(a) + 1`). Although neither Chicory nor eChicory are purity checkers, for this limited purpose we can use a static purity checker (e.g. jPure [16]) to detect cases when either Chicory or eChicory fails to recognize that fields have been updated. When jPure identifies a method as effectful (impure), and only one miner characterizes that effect, we credit that miner with greater accuracy.

4.2 Artifact Selection

We searched GitHub for Java projects (applications and libraries) of size between 2k and 10k lines of code, small enough to control the cost of evaluation but large enough to increase the probability of encountering relevant patterns. We selected projects with indications of popularity ("stars" in GitHub), well-regarded and established development communities (e.g., Apache projects) and, to the extent possible, high test coverage. We also looked at application wikis and issue trackers for mentions of design patterns that could create dynamic relations between clients and library code (observer, model-view-controller, etc).

We selected four artifacts (of which we show two [2], see Table 1), and from these we focused analysis on classes that appeared, from superficial examination, to present interfaces with dynamic creation of relations between client and library code.

4.3 Analysis

To evaluate our approach we ran eChicory and Chicory against each class from Table 1 alone. Ideally, each class should have its own comprehensive suite of unit tests. 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 fed the given traces from eChicory and Chicory to the same version of Daikon. From the resulting invariants we look at whether a method has any effect [3] of any sort (e.g. a variable value is changed) for each given invariant. A method that has reported invariants shows an effect is *non-pure*. Otherwise, the given method appears *pure* based on the given instrumentation technique.

We performed the analysis on a Linux virtual machine with 5GB of memory and an Intel i7-3667U CPU on the host machine.

The only test case provided with `Mockito` [4] has 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 iterate over elements calling methods that eventually change the elements' *state*. *jPure* reports the class has no *pure* methods at all. Both Chicory and eChicory report invariants

[2]We do not show the detailed result for two of the selected artifact as our tests were not fruitful. However, we discuss the reasons in the following section.

[3]Any change on the accessible fields. Including treating a method result as a filed.

indicate that the constructor is not *pure*. Moreover, both agree on showing that the getters are *pure* (all the reported invariants do not show any effect of accessible variables). However, neither Chicory nor eChicory is able to show the effect of the only setter and thus consider it *pure*. eChicory was able to help Daikon infer the effect of the three special purpose methods that were missed by Chicory. These three methods are non-*pure* according to eChicory, which is closer to the result report by *jPure*.

Analysis with the provided test cases was not fruitful for other artifacts. The most common problem was inadequate (ɪ) unit test suites. Some did not cover portions of the library that used dynamic design patterns, and some included broken test cases (most of which we confirmed with the developers e.g. JabRef [3]). A less common issue (ɱ) was mocking frameworks, which made it impossible to observe actual client objects (e.g. Zeppelin [2]). In rare cases (ᴀ) the dynamic relations were never established, for example iterating over an empty list of related objects. Failure due to limitations of our tool (ᴇ) appeared in a single example from Zeppelin [2].

Since provided test suites were inadequate for our purposes, we wrote new test cases for class `DefaultUnknownHandlerManager` from `Apache Struts`. eChicory successfully characterized effects of methods `handleUnknownAction` and `handleUnknownMethod` on other objects, while Chicory found no effects. Our new tests were approved and merged into `Apache Struts`'s main repository.

Performance Impact. While our objective is to improve accuracy and usefulness of analysis, performance cannot be entirely ignored in any tool that, like Daikon, generates and tests conjectures about relations among objects. We can trade some performance for utility, but we cannot afford to forfeit key factors such as the number of variables in scope or the number of defined program points. Our modified technique affects performance in two parts of the overall process, trace collection and invariant inference from traces.

Trace collection on eChicory is followed by trace unification, as explained above. We compare the combined cost of trace collection and unification in eChicory to the cost of trace collection in Chicory. We omitted from our comparison `Struts.ConfigurationManager` since its methods are not implemented and `Zeppelin.Notebook` since we failed to analyze it. The worst cost increase we observed in trace collection and unification was less than 25% (49.79 seconds) for the `Folder` class from Zeppelin [2]; for most classes the difference was negligible. The `Folder` class was also, by far, the worst observed case for the invariant detection step, with a 214% increase in time (from 13.24 to 41.62 seconds); the second worst observed penalty was 40%, and most were much smaller.

The large difference for class `Folder` from Zeppelin [2] is attributable to tracking a large number of dynamically created relations, and suggests that further summarization or selection of representatives may be necessary in some cases. The performance data we have been able to gather so far is quite limited and would not justify any claims of generality, but it at least hints that invariant detection can often be feasible but also that in some cases additional measures will be required to prevent unreasonable costs when large numbers of dynamic relations are created.

5 CONCLUSION

Existing invariant miners dynamically observe values but depend on static associations of variables, making them blind to dynamically established connections between API client and provider. These are not corner cases; they occur frequently in widely used design patterns. We proposed expanding the scope of tracking to encompass dynamic relations such as a client registering with a provider, and produced and evaluated a proof-of-concept implementation.

Evaluation with real-world examples suggests our approach can find invariants missed by current techniques, with reasonable performance costs.

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] 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

[6] 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. https://doi.org/10.1145/1146238.1146267

[7] 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. https://doi.org/10.1145/1368088.1368127

[8] 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. https://doi.org/10.1145/302405.302467

[9] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2382196.2382204

[10] 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. https://doi.org/10.1145/581339.581377

[11] 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. https://doi.org/10.1145/2635868.2635890

[12] 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. https://doi.org/10.1145/2491411.2491457

[13] 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. https://doi.org/10.1145/1368088.1368157

[14] 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. https://doi.org/10.1145/1064978.1065018

[15] 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. https://doi.org/10.1145/2884781.2884873

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

[17] 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. https://doi.org/10.1145/1985793.1985820