# Identifying Inspectors to Mine Models of Object Behavior

Valentin Dallmeier    Andrzej Wasylkowski    Nicolas Bettenburg
Saarland University
Department of Computer Science
Saarbrücken, Germany
{dallmeier,wasylkowski,nicbet}@st.cs.uni-sb.de

## 1 Introduction

In object oriented programming languages, classes are used to incorporate state (fields) and behavior (methods) that modifies the state. Typically, only a subset of a classes' methods actually modifies the state. Methods that don't modify the state are called side–effect free or *pure* methods.

We propose to use purity information to classify methods as *inspectors* (methods that reveal information about an object's state) or *mutators* (methods that change state). An inspector is a pure method that takes no parameters and has a return type other than void. Inspectors can often provide useful abstractions over the internal state of an object. For example, method `isEmpty()` in class `Vector` provides an abstraction over the internal `size` attribute (namely that `size=0`). The benefit of using inspectors is that they provide an abstract characterization of an object's state that does not rely on internal implementation details like fields.

We use inspector and mutator methods to dynamically mine models of object behavior. These models show the effect of mutator invocations on an object's externally visible state (captured by calling all inspectors). Thus, the models are not only meaningful, but also aligned with the view of the user.

In the remainder of this paper we show how common inspector methods are in the wild, and illustrate object behavior models using a real example mined from the execution of the JAVA `Vector` class.

## 2 How Common are Inspectors?

In order to mine object behavior models, we need a sufficiently large number of inspectors. To get an idea how common inspector methods are, we analyzed the purity of eight open-source programs and counted the number of pure methods amongst them.

Our current implementation of the model miner uses the purity analysis provided by Salcianu and Rinard (2005). Their static analysis classifies methods that do not modify objects that existed prior to the invocation as pure, and all others as impure. Unfortunately, the analysis is limited to classes that use only the subset of the JAVA API implemented in GNU ClassPath 0.08 (an open-source implementation of SUN's JDK).

|  | Classes |  | Methods |  |
|---|---|---|---|---|
|  | A | IC | A | I |
| Checkstyle | 83 | 13 | 363 | 38 |
| HTMLParser | 69 | 54 | 469 | 161 |
| JSMSEngine | 26 | 9 | 125 | 27 |
| JackSum | 53 | 34 | 372 | 81 |
| Jalopy | 272 | 146 | 1757 | 225 |
| Lucene | 123 | 62 | 651 | 117 |
| PMD | 194 | 50 | 1397 | 119 |
| ProGuard | 263 | 99 | 3107 | 241 |

Table 1: Number of analyzed (**A**) classes and methods, inspector (**I**) methods and classes with at least one inspector (**IC**).

We searched several sites hosting open-source projects (Apache, SourceForge) and chose eight programs we were able to analyze. The results are summarized in Table 1. It shows the number of analyzed classes and methods as well as the number of inspector methods and the number of classes with at least one inspector.

Our results show that inspector methods occur frequently. On average, half of a program's classes (45%) contain at least one inspector (with the average being 2.44), which is sufficient to investigate the state of an object. The results also reveal that the usage of inspectors differs strongly between projects. While 78% of the classes in `HTMLParser` contain inspectors, this is true for only 15% of the `Checkstyle` classes.

## 3 Object Behavior Models

Having identified inspector methods of a class, we instrument the class by adding invocations to all inspectors before and after each mutator. In order to avoid state space explosion, we abstract from concrete values when converting results of inspector calls to a state (unfortunately, describing this in detail is out of scope of this paper). Executing a program that uses the instrumented class instead of the original one provides us with an object behavior model for that class. This

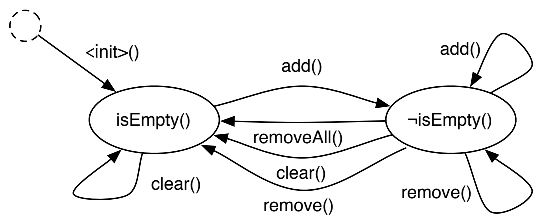

Figure 1: An object behavior model for the JAVA `Vector` class.

process makes our models mirror actual objects' behavior. Like all dynamic analyses, we build on the observation, that *common behavior is often correct behavior*; thus, our models are likely to represent *universal invariants*.

As an example, consider an object behavior model describing the behavior of the JAVA `Vector` class (Figure 1). The model has two states: one, where the vector is empty and one, where the vector holds at least one element. Transitions (and lack thereof) provide us with information about which method calls were, and which were not observed in a particular state during execution and how they changed the state of an object.

We can for instance see, that calling `Vector.add()` always ended up in a non-empty state and that calling `Vector.clear()` always caused the vector to be empty. We can also notice, that a call to `Vector.remove()` was never observed on an empty vector, and this is not without a reason. Calling `Vector.remove()` on an empty vector always fails with an exception being thrown.

On the other hand, some transitions may be non-deterministic. This is the case for `Vector.remove()` method call. Calling this method on a non-empty vector may either change the state of the vector or not. Knowledge in what states a particular method call was observed, and how the call changed that state allows us to extract pre- and postcondition for this method. For example, in case of `Vector.remove()`, it is necessary that the method is called on a non-empty vector, but the call can result in an empty vector as well as a non-empty one.

## 4 Related Work

Usage of finite state automata to abstract behavior of the program was investigated by many researchers in the past years, with Cook and Wolf (1998) being the seminal work about inferring finite state automata from event sequences.

In most approaches, anonymous states have been used (Cook and Wolf, 1998; Ammons et al., 2002). Some researchers have referred to implementation details when labeling states, like in the work of Whaley et al. (2002), who used variables for this purpose.

The work closest to ours is by Xie and Notkin (2004), but, unlike us, they need test cases to generate models and they do not restrict calls to only pure methods when getting the state of the object. Additionally, they do not abstract from concrete values to avoid state space explosion.

## 5 Conclusions and Future Work

Object behavior models capture essential properties of an object from the view of the object's client. Applying partitioning of class' methods into *inspectors* and *mutators*, and using return values of calls to inspectors to represent an object's state distinguishes our work from prior work and is a central contribution of our approach.

As a future work, we plan to use object behavior models to check dynamically, whether executed code does not violate previously learned behavior. We also want to enhance models by extracting state of objects being returned from inspector method calls. This would allow us to express an object's state as the state of its constituents. Another idea is to check programs statically against mined models. Deviations from those models may point us to incorrect usage of an API.

**Acknowledgements** Christian Lindig provided valuable comments on earlier revisions of this paper.

## References

Glenn Ammons, Rastislav Bodík, and Jim Larus. Mining specifications. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.

J. Cook and A. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, number 3385 in LNCS, pages 199–215, January 2005.

John Whaley, Michael Martin, and Monica Lam. Automatic extraction of object-oriented component interfaces. In Phyllis G. Frankl, editor, *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, volume 27(4) of *SOFTWARE ENGINEERING NOTES*, pages 221–231, New York, July 22–24 2002. ACM Press.

Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004)*, pages 290–305, November 2004.