Dynamic Constraint Detection for Polymorphic Behavior *

Nadya Kuzmina Ruben Gamboa
University of Wyoming
{nadya,ruben}@cs.uwyo.edu

Abstract

Dynamic invariant detection, the automatic recovery of partial program specifications by inferring likely constraints from program executions, has been successful in the context of procedural programs. The implementation for dynamic invariant detection examines only the declared type of a variable, lacking many details in the context of object-oriented programs. This paper shows how this technique can be extended to detect invariants of object-oriented programs in the presence of polymorphism by examining the runtime type of a polymorphic variable, which may have different declared and runtime types. We demonstrate the improved accuracy of the dynamically detected specification on two real-world examples: the `Money` example from the JUnit testing framework tutorial, and a database query engine model example, which we adopted from a commercial database application. Polymorphic constraints in both cases are shown to reveal the specification of the runtime behavior of the systems.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Class invariants—automatic specification recovery

General Terms Verification, Algorithms

Keywords object constraints, class invariants, dynamic invariant detection, automatic specification recovery

1. Introduction

A constraint[1] is a restriction on one or more values of (a part of) an object-oriented model or system [5]. Constraints are checked by `assert` statements at runtime to guarantee that desired properties hold. Constraints on visual formal models, such as class diagrams, provide for better documentation, improved precision, and allow communication with fewer misunderstandings among team members.

Dynamic invariant detection automatically generates *likely* constraints by examining program executions. *Likely* constraints are properties that hold on the examined program runs [4].

* The material is based upon work supported by the National Science Foundation under Grant No. 0613919.

[1] A constraint is called "invariant" in the Daikon literature. We are using the term "constraint" to refer to properties of object-oriented, as opposed to procedural, systems.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

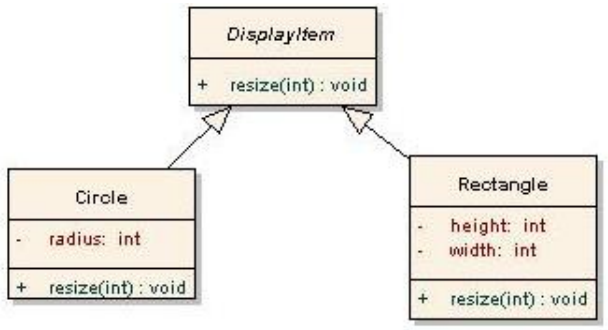

Figure 1. `DisplayItem` Example Class Diagram

Polymorphism is one of the challenges constraint detection tools need to address in object-oriented programs. The declared type of a polymorphic variable may not fully characterize the variable's behavior. For example, let `Circle` and `Rectangle` be the child classes of an abstract class `DisplayItem`[2] which has no declared fields, as shown on figure 1. The `resize(int amount)` method of a single `DisplayItem` variable will scale the `radius` when applied to a `Circle` instance and the `width` and `height` when applied to a `Rectangle` instance, but we need to examine the fields of the different runtime instances in the `DisplayItem` variable to state these properties about the `resize` method.

Our approach demonstrates the feasibility of characterizing polymorphic behavior by inferring polymorphic constraints for different runtime classes in Java. Our prototype implementation, Turnip, considers the fields of runtime polymorphic variables in a way that yields runtime-refined polymorphic constraints. Such constraints have the form of an implication where the antecedent specifies a particular runtime class and the consequent is a constraint on the fields of the class. Consider the `DisplayItem` example. Suppose the application is displaying a complex graphical component which contains a `DisplayItem figure` as its attribute. The user decides to adjust the size of the component by `amount`, which causes the component to resize the `figure` and redraw itself. Some of the postconditions of the `redraw` method of the component may be as follows:

```
( figure.class == Circle ) ==>
( figure.radius == figure.radius@pre * amount )
( figure.class == Rectangle ) ==>
( figure.width == figure.width@pre * amount ).
```

In the rest of the paper, we discuss a problem with dynamic invariant detection in an object-oriented setting, present our solution and list its limitations.

[2] This example is inspired by [2].

2. Dynamic Invariant Detection with Daikon

Daikon [1, 4], developed by Michael Ernst and his research group, is a general and publicly available implementation for dynamic invariant detection. Constraints are captured as "operational abstractions" which are the types of formulas programmers may place in `assert` statements, such as $x >= 0$ or $y = x * z$. Constraints are stated in terms of variables present in a program.

Being a general purpose tool for a variety of languages, Daikon does not provide specific object-oriented support for polymorphism and inheritance. Daikon only considers the fields that are guaranteed to be present, given the declared type of an object. The problem arises when the declared type of a polymorphic variable is a Java interface, which has no fields, or a superclass with relatively few fields. The absence of fields results in the inability of Daikon to infer many constraints characterizing the behavior of such polymorphic variables.

Daikon previously provided a less satisfactory solution to capturing runtime behavior known as a runtime-refined types mechanism in the older, deprecated front end for Java, `dfej`. `dfej` allowed to refine the runtime type of a polymorphic variable to one specific runtime type via an annotation. For example,
`/*refined_type: Integer*/ Object element;`
makes `dfej` treat `element` as a variable of class `Integer`. However, the annotation mechanism does not account for polymorphic cases when it is impossible to limit the runtime class of a variable to only one particular class, which is quite often the purpose of using polymorphism.

3. Our Approach

We call our version of Daikon augmented to consider runtime-refined cases Turnip, in accordance with the naming scheme devised by Daikon developers[3]. Turnip examines the fields of runtime objects to identify runtime variable values to identify the constraints that likely hold between them. In the presence of polymorphism, Turnip examines the actual runtime class of each program variable to infer properties that likely hold for the fields in the examined runtime class. With polymorphism, examining variables specific to the actual runtime class of a declared program variable yields properties that are likely to hold for the variables in the examined runtime class. We call such properties runtime-refined constraints. For example, consider the `DisplayItem figure` variable. Examining the values of the `figure.radius` attribute when the runtime class of `figure` is `Circle` yields relationships between `figure.radius` and other visible variables at a particular program point.

Runtime-refined constraints reveal the characteristic behavior for the known classes of a polymorphic variable. These details remain hidden due to the lack of examined subclass fields if only the declared type of a polymorphic variable is taken into account.

3.1 Modifications to Daikon

Chicory is the front end for Daikon that instruments Java classes when they are loaded by the JVM with the purpose of collecting the runtime values of the variables.

We enabled Chicory to collect values for the fields of the actual runtime classes of each polymorphic variable. Chicory has an inner representation for different kinds of variables in a program. We introduced a new variable abstraction into Chicory, called a *group variable*, which represents polymorphic variables declared in the target program. The program variable that underlies a particular group variable is referenced as its *base variable*. A polymorphic variable is a variable declared as a user-defined class that has child classes. The value of such variable can be an object of the declared parent class (if it is not abstract) or an object of one of the child classes. In the `DisplayItem` example, a variable declared as `DisplayItem figure` is a polymorphic variable. The group variable's implementation is based on the state design pattern [3], allowing it to alter its behavior at run-time when the runtime class of the base variable changes. The state of a group variable represents the current runtime class of its base variable and is changed every time the base variable changes its runtime class. A group variable then delegates all value collecting activity to the current state object. We also introduced a similar mechanism for reading in values for polymorphic variables into Daikon.

[3] Daikon is an Asian radish.

4. Limitations

Constructing extensive hierarchies is prohibitive in terms of used resources. This limitation can be overcome by disregarding Chicory's assumption that all variables have an a priori known fixed type. Then only actual runtime types would be considered for polymorphic variables.

Turnip processes more variables per program point than Daikon does, which results in decreased performance and more accidental properties reported by Turnip. This problem is related to the nature of dynamic constraint detection. It can be partially solved by disabling some properties, and, perhaps, adjusting the statistical justification threshold. Such fine-tuning mechanisms are built into Daikon.

More relevant invariants can be produced by combining static program analysis techniques with dynamic detection. Symbolic evaluation can be used to augment dynamic analysis with the knowledge of underlying source code. Abstract interpretation might aid in pruning the search space of potential properties for dynamic analysis.

5. Experiments

We validated our approach on two real world examples: the `Money` example from the JUnit testing framework tutorial, and a database query engine model example, which we adopted from a commercial database application. Both cases offer an insight into the behavior of the system that Daikon alone could not.

6. Conclusions

We have demonstrated that examining polymorphic behavior results in better accuracy of dynamically inferred specifications for object-oriented systems. Our prototype implementation for dynamic invariant detection with runtime-refined cases, built upon Daikon, produced compelling results for two real world systems.

References

[1] Daikon invariant detector. `http://pag.csail.mit.edu/daikon`.

[2] G. Booch. *Object-oriented analysis and design with applications*. Benjamin-Cummings, second edition, 1994.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[4] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, November 2–4, 2004.

[5] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.