Extending Dynamic Constraint Detection with Polymorphic Analysis

Nadya Kuzmina and Ruben Gamboa*
University of Wyoming
P.O. Box 3315
Laramie, WY 82071-3315
{nadya, ruben}@cs.uwyo.edu

Abstract

The general technique for dynamically detecting likely invariants, as implemented by Daikon, lacks specific object-oriented support for polymorphism. Daikon examines only the declared type of a variable which prohibits it from examination of the runtime variables in the presence of polymorphism. The approach presented in this paper extends the technique to consider the runtime type of a polymorphic variable, which may have different declared and runtime types. The runtime behavior of a polymorphic variable is captured by polymorphic constraints *which have the form of an implication with the name of the runtime class in the antecedent. We demonstrate the improved accuracy of the dynamically detected specification on the* `Money` *example from the JUnit testing framework tutorial. Polymorphic constraints are shown to reveal the specification of the runtime behavior of the example.*

1. Introduction

Our goal is to recover polymorphic constraints[1] that provide insight into the runtime behavior of an object-oriented system. Such constraints will aid developers in understanding object-oriented code better. Turnip, our prototype implementation for polymorphic constraint detection, is based on Daikon (version 4.1.6), a dynamic invariant detection tool developed with similar goals in mind.

Polymorphism is one of the challenges constraint detection tools need to address in object-oriented programs [11]. While true in procedural programming, the assumption that the declared type of a variable defines its actual runtime behavior does not hold for object-oriented systems where inheritance and polymorphism come into play. Polymorphic behavior requires examining the actual runtime type of a program variable to grasp meaningful constraints on it.

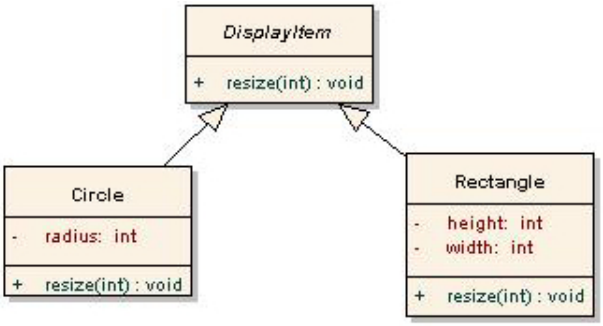

Figure 1. `DisplayItem` **Example Class Diagram**

The declared type of a polymorphic variable may not fully characterize the variable's behavior. For example, let `Circle` and `Rectangle` be the subclasses 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

*This material is based upon work supported by the National Science Foundation under Grant No. NSF CNS-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.

[2] This example is inspired by [5].

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. 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 )
```

The rest of this paper is organized as follows. The next section describes the background on dynamic invariant detection and attempted solutions related to our work. Section 3 provides details on our approach and concludes with its limitations. Section 4 presents and compares constraints detected by Daikon and Turnip[3] (our augmented version of Daikon) on the `Money` example from the JUnit testing framework tutorial. Finally, section 5 presents our conclusions.

2. Background

Daikon [1], developed by Michael Ernst and his research group, is a general and publicly available implementation for dynamic invariant detection which captures constraints as operational abstractions. A full description of Daikon can be found in [9, 13].

Being a general purpose tool for a variety of languages, Daikon does not provide specific object-oriented support for polymorphism and inheritance. Daikon considers only the declared type of a variable when instantiating properties. In the `DisplayItem` example, Daikon uses variables declared in the target program to construct properties to be checked. Given the declaration `DisplayItem figure`, Daikon will not instantiate any properties on the fields of `figure` because the `DisplayItem` class has no declared fields. On the other hand, if we consider the actual runtime class of `figure`, the `figure.radius` field of the `Circle` class, and `figure.width` and `figure.height` fields of the `Rectangle` class can be used to instantiate properties.

Daikon used to provide a rudimentary solution to capturing runtime behavior at the level of the Java front end. It is known as the runtime-refined types mechanism in Daikon's older, deprecated front end for Java, `dfej`. Under the assumption that a variable's runtime value can be guaranteed to be of one specific type, `dfej` allowed an annotation specifying the refined type to be put before the corresponding variable declaration as in */*refined_type: Rectangle*/ DisplayItem figure;* `dfej` would then treat `figure` as a variable of class `Rectangle`. By the definition of polymorphism [4], the classes in one hierarchy are interchangeable. The actual class of a variable is determined by the runtime context and is not known beforehand. 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.

Another attempt to solve the problem that many classes do not have fields that an invariant detector can examine is presented by Daikon's provision of *pure methods*. A *pure method* is a read-only method which returns a value. However, this approach does not provide access to the fields of the runtime class which often participate in desirable constraints. Therefore, the combination of pure methods with runtime fields is most beneficial. More details are presented in [12].

Csallner and Smaragdakis [6] explore the problem of eliminating invariants that are inconsistent with the behavioral subtyping principle for overriding methods in object-oriented programs with the purpose of using the invariants in automating reasoning tasks. In contrast, our approach infers constraints on runtime types of a polymorphic variable in an object-oriented system so that a human developer gets a more accurate representation of the system's runtime behavior.

3. Dynamic Detection of Polymorphic Behavior

In this section we present our approach to inferring polymorphic behavior from data traces and its limitations.

3.1. Our Approach

Turnip examines the fields of runtime objects to identify runtime variable values in order to infer 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. 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.

Chicory is the front end for Daikon that instruments Java classes when they are loaded by the JVM. Before the JVM loads a class, Chicory creates a tree[4] of the visible variables at each program point. Each node in the tree represents a

[3] To preserve the naming scheme of Daikon developers.

[4] A tree is a good data structure that reflects the recursive nature of variable nestings. For instance, the fields of an object take other objects as values which expose their own fields, and so on.

0-7695-2963-1/07 $20.00

variable in the target program and is responsible for retrieving the runtime value of this variable.

After the tree is constructed, the JVM loads the class and executes the target program. Each time the execution gets to a program point, Chicory traverses the variable tree for that program point and collects variable values from each node. The variable values are stored into a data trace file for further processing by Daikon.

We enabled Chicory to collect values for the fields of the actual runtime classes of each polymorphic variable. The system identifies a polymorphic variable as a variable declared as a user-defined interface or a user-defined class that has subclasses. The value of such a variable can be an object of the declared parent class (if it is not abstract) or an object of one of the subclasses. In the `DisplayItem` example, Turnip correctly identifies a variable declared as `DisplayItem figure` as a polymorphic variable. The way we implement it in Chicory is to maintain a collection of class hierarchies based on the classes that have been loaded by the JVM. Every time a subclass is being loaded, it is added to the appropriate hierarchy tree by Chicory. This mechanism allows Chicory to identify polymorphic variables as variables whose declared type is a member of one of the hierarchies and is not a leaf.

Chicory uses variable abstraction to represent different kinds of variables in a program. We introduced a new variable abstraction, called a *group variable*, to represent polymorphic variables declared in the target program. The program variable that underlies a particular group variable is referred to as its *base variable*.

The group variable's implementation is based on the state design pattern [10], 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 modified the data collecting mechanism in Chicory. During runtime, Chicory records the values observed for a particular program point into a trace file, which consists of two parts: the declarations part and the actual data part. The declarations part contains declarations for all variables present at the specified program point. The data part records variable values observed during execution for each program point. For a polymorphic variable, the declarations part lists all fields that this variable could possibly have for different runtime classes. The data part records only the runtime class of a polymorphic variable and the values observed for the fields of the runtime class.

We introduced a similar mechanism for reading in values for polymorphic variables into Daikon's backend. This mechanism allows us to reuse the statistical justification mechanism built into Daikon for polymorphic constraints. To account for the fact that a variable can be of only one class at any given time, we prohibit the construction of properties that result from combining fields of different runtime classes of the same base variable. Thus for `DisplayItem figure`, we prohibit property construction on both `figure.radius` and `figure.width` variables (such as `figure.radius > figure.width`, for example).

3.2. Comparative Analysis of Daikon and Turnip

In this section we first analyze the space and time complexities of Turnip compared to Daikon [13]. Next we argue that our modifications do not result in the loss of constraints given the same number of samples per program point. More detailed analysis are provided in [12].

Let us consider a polymorphic variable *A a*. Let *n* be the number of classes in the inheritance hierarchy of class *A*, and let *m* be the maximum number of fields for classes in the *A*-hierarchy. The number of variables that Turnip considers for *A a* is upper bounded by $n*m$. Therefore, there is only a linear increase on the number of variables considered by Turnip compared to the number of variables considered by Daikon. Daikon's analysis [13] suggests that the number of potential constraints is cubic in the number of variables in scope at a program point because constraints in Daikon involve at most three variables. Thus in the worst case the space complexity of Turnip is cubic in the size of the largest inheritance hierarchy in a program. This is a crude upper bound because in practice Turnip considers only the types of a polymorphic variable that are observed at runtime as opposed to the complete static inheritance hierarchy of the variable. Also, as the Daikon community [8] suggests, most of the constraints are falsified quickly and only a small number of constraints that are never falsified need to be checked for all samples, so the constraint detection time is really linear in the small number of never-falsified constraints.

Another concern may be that Turnip fails to infer some constraints that Daikon alone does. A simple argument shows that this is not the case. Since Turnip observes the superset of variables observed by Daikon, constraints involving just the common variables will be observed with the same frequency in Turnip as in Daikon. However, more trials will be required to obtain the statistical significance of polymorphic constraints.

3.3. Limitations

At the moment Turnip conceptually assumes that all inherited fields are used for specialized purposes in the subclasses. However, some inherited fields may serve the same

purpose in all subclasses. In this case a postprocessing step may be used to propagate the constraints that conceptually belong to the superclass from its subclasses. For example, suppose class A has a field a and subclasses B and C. If $Q(a)$ holds for the runtime class B as well as for the runtime class C then generalize that $Q(a)$ holds for the runtime class A.

Extensive hierarchies are prohibitive in terms of used resources since the number of potential constraints is cubic in the size of a hierarchy, as discussed in 3.2. Therefore, our approach considers only user-defined hierarchies of classes. For example, we are not refining variables declared as, say `Object`.

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.

4. Extended Examples

In this section we describe the *Money* and the *Database Query Engine Model* examples which offer an insight into the behavior of the systems that Daikon alone could not.

The JUnit [2] framework documentation contains the following example to help developers get started with writing unit test cases. We will use this example to demonstrate the improved precision in recovered specifications obtained by introducing runtime-refined cases. The example represents arithmetic with multiple currencies. The system consists of an interface `IMoney` and two classes, `Money` and `MoneyBag`, implementing the `IMoney` interface, as presented in figure 2.

The `Money` class represents a quantity of money in a particular currency. The amount is represented by a simple `int` field `amount`. The currency is represented by a string holding the ISO three letter abbreviation (e.g., "USD" and "CHF"). The `MoneyBag` class stores different monies in the `Vector fMonies` field.

To accommodate the logic of the `Money` example we introduced a derived variable, the sum of all elements in a `java.util.List` collection, into both Daikon and Turnip. The `sum` is created if and only if there is a Java annotation present before a `java.util.List` variable declaration in the source code as in:

```
@daikon.chicory.ListOfNumbers
Vector fMonies;
```

To accommodate the `sum`, non-number classes are required to return their numeric representation from the `toNumber()` method.

The `Money` example came with a set of unit tests. In the experiment Daikon and Turnip were presented with the respective outputs from the same runs of each unit test. The summary of constraints that characterize the behavior of the `Money` and `MoneyBag` classes with results for Daikon and Turnip is presented in figures 3 and 4.

Before we turn to the results in figures 3 and 4, let us consider the `IMoney add(IMoney m)` method in the `MoneyBag` class which adds a money `m` to the current money (`this` object). It presents an interesting case with two polymorphic variables: the input parameter `m` and the `return` object. Since both can have a runtime class of either `Money` or `MoneyBag`, there are four runtime-refined cases which Turnip outputs:

```
IMoney MoneyBag.add(IMoney m):::EXIT
// Constraint inferred by Daikon
return != null
// runtime-refined constraints inferred by Turnip
( m.class == MoneyBag & return.class == MoneyBag ) ==>
( m.fMonies[].sum -
    return.fMonies[].sum + this.fMonies[].sum == 0 )
( m.class == MoneyBag & return.class == Money ) ==>
( m.fMonies[].sum -
    return.fAmount + this.fMonies[].sum == 0 )
( m.class == Money & return.class == MoneyBag ) ==>
( m.fAmount -
    return.fMonies[].sum + this.fMonies[].sum == 0 )
( m.class == Money & return.class == Money ) ==>
( m.fAmount - return.fAmount + this.fMonies[].sum == 0 )
```

All the runtime classes reported by Turnip for the `IMoney` variables were encountered during the `Money` example execution with the test cases provided. Turnip uses dynamic analysis of variable values encountered during target program runs, making it impossible to infer properties about the runtime classes that have not been seen during the examined execution.

Written by Kent Beck and Erich Gamma, the implementation for the `add` method in `Money` and `MoneyBag` employs the double dispatch pattern, which means that the method returns a new `IMoney` object, without modifying the current one (`this` object). Turnip examined the polymorphic behavior of both the input parameter `m` and the `return` object, and the fields of `this` to produce the runtime-refined constraints which reflect the fact that the `add` method returns an `IMoney` that is the addition of the money in the input parameter `m` to the current money. Daikon, on the other hand, skips objects declared as interfaces using the assumption that interfaces do not have fields and examines only the fields of the current object when inferring constraints. With the double dispatch, the fields of the current object do not change. Therefore, Daikon is not able to infer any constraints for the `add` method except the comparison of the returned object[5] to `null`.

Figures 3 and 4 present constraints that characterize the polymorphic behavior of the following methods from the

[5]The property `m != null` only appears at the entry point to a method, and is not reported at the exit point.

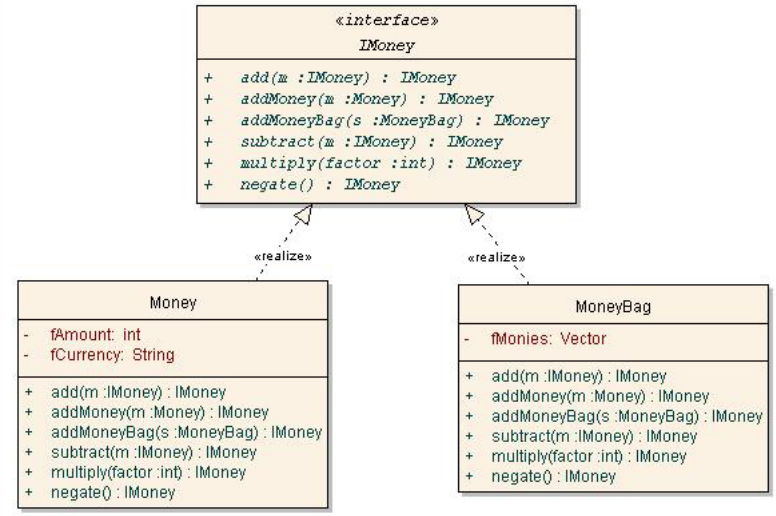

Figure 2. `Money` **Example Class Diagram**

Method	*Postcondition*	*Daikon*	*Turnip*
IMoney add(IMoney m)	(m.class == MoneyBag & return.class == MoneyBag) ⇒ m.fMonies[].sum - return.fMonies[].sum + this.fAmount == 0	–	√
	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum - return.fAmount + this.fAmount == 0)	–	√
	(m.class == Money & return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fAmount == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fAmount == 0)	–	√
IMoney addMoney(Money m)	(return.class == Money) ⇒ (m.fCurrency == return.fCurrency)	√	√
	(return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fAmount == 0)	√	√
	(return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fAmount == 0)	√	√
IMoney addMoneyBag(MoneyBag s)	(return.class == MoneyBag) ⇒ (s.fMonies[].sum - return.fMonies[].sum + this.fAmount == 0)	–	√
	(return.class == Money) ⇒ (s.fMonies[].sum - return.fAmount + this.fAmount == 0)	–	√
IMoney subtract(IMoney m)	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum + return.fAmount - this.fAmount == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount + return.fAmount - this.fAmount == 0)	–	√
IMoney multiply(int factor)	(return.class == Money) ⇒ (return.fAmount == (this.fAmount * factor))	–	√
IMoney negate()	(return.class == Money) ⇒ (return.fAmount == – this.fAmount)	–	√

Figure 3. Postconditions which reflect the behavior of `IMoney` interface methods implemented in the `Money` class. `return.fMonies[].sum` stands for the sum over the numerical representation of all members of array `fMonies[]`. Checkmark (√) means that the corresponding constraint was detected, dash (–) means that the corresponding constraint was not detected.

0-7695-2963-1/07 $20.00

Method	*Postcondition*	*Daikon*	*Turnip*
IMoney add(IMoney m)	(m.class == MoneyBag & return.class == MoneyBag) ⇒ (m.fMonies[].sum - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum - return.fAmount + this.fMonies[].sum == 0)	–	√
	(m.class == Money & return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fMonies[].sum == 0)	–	√
IMoney addMoney(Money m)	(return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fMonies[].sum == 0)	–	√
IMoney addMoneyBag(MoneyBag s)	(return.class == MoneyBag) ⇒ (s.fMonies[].sum - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(return.class == Money) ⇒ (s.fMonies[].sum - return.fAmount + this.fMonies[].sum == 0)	–	√
IMoney subtract(IMoney m)	(m.class == MoneyBag & return.class == MoneyBag) ⇒ (m.fMonies[].sum + return.fMonies[].sum - this.fMonies[].sum == 0)	–	√
	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum + return.fAmount - this.fMonies[].sum == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount + return.fAmount - this.fMonies[].sum == 0)	–	√
IMoney multiply(int factor)	(return.class == MoneyBag) ⇒ (return.fMonies[].sum == (this.fMonies[].sum * factor))	–	√
IMoney negate()	(return.class == MoneyBag) ⇒ (return.fMonies[].sum == – this.fMonies[].sum)	–	√

Figure 4. Postconditions which reflect the behavior of `IMoney` interface methods implemented in the `MoneyBag` class. `return.fMonies[].sum` stands for the sum over the numerical representation of all members of array `fMonies[]`.

`IMoney` interface in classes `Money` and `MoneyBag` respectively:

`IMoney add(IMoney m)` Adds a money `m` to this money.
`IMoney addMoney(Money m)` Adds a simple `Money` to this money.
`IMoney addMoneyBag(MoneyBag s)` Adds a `MoneyBag` to this money.
`IMoney subtract(IMoney m)` Subtracts a money `m` from this money.
`IMoney multiply(int factor)` Multiplies `this` money by `factor`.
`IMoney negate()` Negates `this` money.

Turnip inferred its constraints by examining the runtime classes of polymorphic variables. Daikon was not able to infer most of the constraints because it only examined the declared type of variables, ignoring the actual runtime classes of polymorphic variables.

Even though Daikon did not infer the specified constraints in our experiments for the `multiply` and `negate` methods, `dfej`, the older front-end for Daikon described in section 2, should have been able to detect them with an appropriate annotation in the source code. It is possible to insert `dfej`'s annotation specifying the runtime class of the returned objects for these two methods, because they always return objects of the class in which they are declared (e.g., `negate()` in the `Money` class always returns an object of runtime class `Money`).

Daikon successfully inferred polymorphic constraints for the `addMoney` method in the `Money` class. The success in this particular case is explained by the dynamic checks of method returns which are built into Daikon [7]. The implementation for the `IMoney addMoney(Money m)` method in class `Money` returns a `Money` object if `m` contains the same currency as the current money, and a `MoneyBag` object in the other case. Daikon looks for different behavior in multiple `return` statements, in this case the two `return` statements differ by the runtime class, enabling Daikon to infer polymorphic constraints. In general, method return analysis in Daikon is not runtime class specific and may not be able to produce polymorphic constraints in a more complicated case.

The `Money` example makes heavy use of polymorphism which results in the increase of the running time used by Turnip compared to Daikon to infer the constraints. Turnip takes about twice as much time to infer constraints for the `Money` example as Daikon (33.2 seconds for Turnip, 17.4 seconds for Daikon).

Let us note that although the runtime-refined cases produced by Turnip convey the specification for the `IMoney add(IMoney m)` method, the `sum` derived variable does not distinguish between different currencies. For example, if we add a `MoneyBag b`, which contains 5 USD and 3 CHF, to a `Money m`, which represents 7 USD, the amount of USD in the returned `MoneyBag r` is the sum of the amount of USD in `b` and `m`, which is 12 USD. The amount of CHF in `r` is equal to the amount of CHF in `m` and does not get added to anything. The constraint specified via the `sum` derived variable for

this case is `m.fMonies[].sum + this.fAmount == return.fMonies[].sum`, which does not reflect that the addition occurred only with USD, but not with CHF. Constraints involving the `sum` derived variable can only specify relationships with the total sum of all currencies in a `MoneyBag`. A proper constraint relating the amount of money in a particular currency `X` in the resulting `MoneyBag return` with the amount of money in currency `X` in the input `MoneyBag m` and the amount of money in currency `X` in the current `Money (this)` is as follows (stated in OCL):

(m.class == MoneyBag & return.class == MoneyBag) ==>
(return.fMonies[]->select(fCurrency == this.fCurrency).fAmount ==
this.fAmount + m.fMonies[]->select(fCurrency==this.fCurrency).fAmount)

Such constraints are too complex for Daikon's current dynamic detection technique.

We also verified Turnip on a model that we extracted from the query engine of a production quality database system MIM [3]. MIM provides support for various financial queries. Our test queries included such queries as "SHOW (Low of IBM + High of IBM) * 0.5" and "SHOW Close of IBM + Return of IBM".

The query engine creates an executable query based on a parsed request represented as a tree of nodes. Each node is capable of returning its value for a particular date. We modeled nodes which support addition, multiplication and negation of base entities which can be of two types: a constant and a relation column. Each tree node inherits from the abstract class `ExecNode` which declares the `getValue(int idx)` method whose purpose is to return the value of the current object at the specified date represented by `idx`. The classes in the `ExecNode` hierarchy override the `getValue(int idx)` method to return the appropriate value on the specified date.

Both Daikon and Turnip detected the characteristic constraints for the base entities because there is no need for polymorphism in the case of the base entities. Polymorphic variables were used for the tree nodes representing the operations of addition, multiplication and negation. Turnip, but not Daikon, succeeded in specifying the behavior of the `getValue(int idx)` method for each of the operations. A full description of this example is available in [12].

5. Conclusions

Our experiments suggest 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 non-trivial systems. Both runtime-refined specifications offer more precision and insight into the behavior of the underlying system than the corresponding specifications without polymorphic cases.

The limitations and drawbacks of the prototype implementation define our future work. The "noise" (reported accidental or irrelevant properties) in Daikon's output suggests the use of static program analysis techniques to improve the quality of the reported constraints. Using symbolic evaluation to gain knowledge of the underlying source code and abstract interpretation to narrow the search space for dynamic analysis are promising lines of research. We are currently working on a version of a tool that may be able to dynamically detect OCL-specified constraints [14] in object-oriented systems.

References

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

[2] Junit. `http://www.junit.org`.

[3] Xmim database server. `http://www.lim.com`.

[4] E. Baude. *Software Design: From Programming to Architecture*. Wiley, 2004.

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

[6] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *Proc. 28th International Conference on Software Engineering, Emerging Results Track*, pages 861–864, May 2006.

[7] N. Dodoo, A. Donovan, L. Lin, and M. Ernst. Selecting predicates for implications in program analysis, 2002.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

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

[11] N. Kuzmina and R. Gamboa. Dynamic constraint detection for polymorphic behavior. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006), Poster Track*, Portland, OR, USA, October 22–26, 2006.

[12] N. Kuzmina and R. Gamboa. Extending dynamic constraint detection with polymorphic analysis. Technical report, University of Wyoming Department of Computer Science technical report UWCS-07-01 (Laramie, WY), January 2007.

[13] 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, Newport Beach, CA, USA, November 2–4, 2004.

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

0-7695-2963-1/07 $20.00 © 2007 IEEE