Extending Dynamic Constraint Detection with Polymorphic Analysis

Nadya Kuzmina, 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, implemented by Daikon, lacks specific object-oriented support for polymorphism. Daikon examines only the declared type of a variable which prohibits it from considering 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 two 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.*

1. Introduction

A constraint[1] is a restriction on one or more values of (a part of) an object-oriented model or system [26]. 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 [23]. Dynamic invariant detection has been applied to a variety of practical problems, such as automating theorem proving tasks [19, 20], verifying safety properties [21, 22], generating and prioritizing test cases [28], program refactoring [16], and error detection [25, 24] among others. In contrast to using constraints as input to other tools, we focus on providing quality constraints to developers.

There is a variety of tools which we describe in section 2 that attempt to discover program models or specifications using dynamic analysis. Our goal is to recover polymorphic constraints that provide insight into the runtime behavior of an object-oriented system in terms of program variables. 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.

In the current object-oriented paradigm Object Constraint Language (OCL) [26] is the standard set by the Object Management Group for specifying constraints for object-oriented systems. OCL allows the developer to specify both functional (e.g, $x \geq 0$) and object-oriented (e.g., *self.oclType* = *Rectangle* for a *Rectangle*) relationships. Tools that automatically detect object-oriented constraints and present them in OCL will improve the readability and precision of the recovered specifications for object-oriented systems.

Polymorphism is one of the challenges constraint detection tools need to address in object-oriented programs [17]. 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. In terms of dynamic invariant detection, polymorphic behavior requires examining the actual runtime type of a program variable to grasp meaningful constraints.

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

[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.

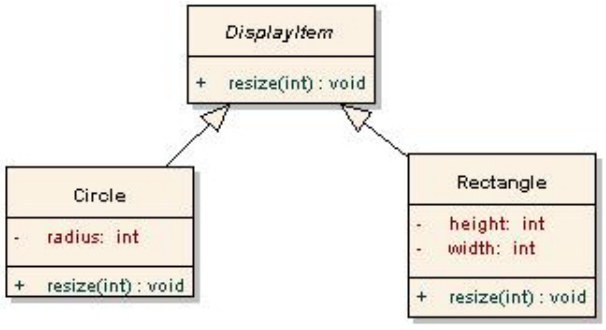

Figure 1. `DisplayItem` **Example Class Diagram**

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. 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 )
( figure.class == Rectangle ) ==>
  ( figure.height == figure.height@pre * amount )
```

The rest of this paper is organized as follows. The next section describes the background and related work on dynamic invariant detection. 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 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 [3]. Finally, section 5 presents our conclusions.

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

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

2. Background

We start by providing some background on Daikon and give a brief description of other research based on dynamic program analysis.

2.1. Daikon

Daikon [1, 23], 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 $y = x * z$.

Daikon consists of two separate and mutually independent parts. Daikon's language-dependent front end instruments the target program to trace variables at certain locations in the program. The front end collects variable values into a data trace file during the target program's execution over a test suite. Daikon supports a variety of front ends for both object-oriented and procedural languages, including Chicory, a front end for Java. Daikon's language-independent back end examines the data trace file to infer constraints on the variables. The distinction between these two parts has blurred in the latest versions of Daikon, but it is still best understood with this separation in mind.

The constraints inferred by Daikon are determined by *program points*, a *grammar of properties*, and the variables visible at the various program points. *Program points* are locations in the target program where constraints should be inferred. For example, method entry and exit locations correspond to method preconditions and postconditions and are natural candidates for program points. Daikon's front ends produce a data trace file that contains state variable values at each program point.

A fixed *grammar of properties* is a list of templates that describe possible relationships between variables. For example, one template may look like $\alpha >= 0$ and another $\alpha == \beta * \gamma$, where the metavariables α, β and γ are typed. Each such template is instantiated with all possible combinations of the program variables of correct type visible at a program point. Suppose that *int x, int y, int z* are visible at program point P_1. Then, the first template will be instantiated as $x >= 0$, $y >= 0$, and $z >= 0$, and the second template will be similarly instantiated as $x == y * z$, $y == x * z$, $z == y * x$, $y == z * x$, $z == x * y$, and $x == z * y$. After templates have been instantiated, Daikon checks to see if the variable values from each data trace

for program point P_1 satisfy each instantiated template at P_1. In the end Daikon reports the *likely* constraints as the properties that are never invalidated by any data trace. We will use the term *property* to denote a template from the grammar of properties that has been instantiated over the variables at a program point.

Daikon outputs two kinds of *likely* constraints: accidental properties and essential constraints. The latter are constraints in the traditional sense, whereas accidental properties are an artifact of the values observed during the examined executions and are not universally true for all program runs.

Daikon attempts to minimize the number of reported constraints. First, it uses statistical justification to distinguish chance relationships from likely constraints. Daikon establishes the properties that hold on the given data trace, and then for each property, it computes the probability that the observed property could have happened by chance alone on a random set of samples. Only properties whose probability is smaller than the user-defined confidence parameter qualify as likely constraints and are reported. Second, Daikon suppresses constraints that are easily derived from one that is reported. Although incomplete, the description of Daikon offered here is enough for our purposes. A full description can be found in [23, 10].

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 following 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.

There are two mechanisms that attempt a solution to the problem that many classes do not have fields that an invariant detector can examine. The first mechanism allowed the invariant detector to explore fixed runtime types. The other mechanism provides more variable for the invariant detector to examine.

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 provide more variables to the invariant detector is present in the current version of Daikon as a way to use a *pure method*, a read-only method which returns a value, as a derived variable. The return type of a pure method becomes the type of the associated derived variable. From the point of view of the reported invariants, a derived variable is equivalent to any "regular" variable which is in scope at a program point. This mechanism attempts a solution to the problem that many classes do not have fields that an invariant detector can examine. In some very specific cases this approach may offer a solution for polymorphic variables. For example, suppose that `DisplayItem` has a pure method `double area()`. Suppose also that the program point `A` has a `DisplayItem figure` variable, which is specified to always have the area of, say, 1.0. In this case, the pure method mechanism allows Daikon to detect the desirable constraint on the polymorphic variable `figure`: `figure.area() == 1.0`. However, in a general case, the fields of the runtime class often participate in desirable constraints. For instance, in the example with the `area` method, the desirable constraint may be the one that provides the formula for the area, such as:

```
( figure.class == Rectangle ) ==>
    ( area() == figure.height * figure.width )
```

This example demostrates that pure methods and runtime fields are a beneficial combination for dynamic invariant detection.

2.2. Time and Space Complexities for Dynamic Constraint Detection

This section considers the approximate time and space costs of dynamic constraint detection based on an incremental (single-pass) algorithm [23] which discards each sample after processing it. The material in this section was first presented in [23]. We briefly describe the complexity analysis of Daikon to provide a framework for examining Turnip's space and time costs.

The algorithm consists of three steps, which treat each program point independently:

1. For each program point, create all possible candidate constraints by instantiating all appropriate templates

V	total number of variables at a program point
L	execution length: number of samples for a program point
C	number of possible constraints at a program point = *G(V)*
RC	number of reported constraints at a program point
P	program size: number of program points
G(v)	grammar: number of constraint templates, given *v* variables.

Figure 2. Variables used in the time and space complexity analysis.

from the *grammar of properties* with the variables at the program point.

2. For each sample at each program point, check all candidate constraints and discard the ones that are falsified by the given sample.

3. Report the constraints that remain after processing all samples and applying the post-processing filtering.

The algorithm uses space only to store candidate constraints. The maximum space usage which occurs during the initial step is $S = O(P * C) = O(P * G(V))$. The worst-case runtime scenario occurs in the initial phase when the constraint detector has to check all candidate constraints for each sample: $T = O(P*C*L)$. Since most of the candidate constraints are falsified quickly, the common-case time complexity is $T = O(P * C + P * RC * L)$, where $P * C$ stands for falsified properties and $P*RC*L$ denotes the reported properties, which are checked for all samples. More details on time and space complexity analysis for simple incremental algorithm and Daikon can be found in [23, 9].

Daikon uses bottom-up incremental algorithm which improves on the simple incremental algorithm by orders of magnitude by introducing serveral optimizations [23]. The simple incremental algorithm provides a good conceptual picture for time and space complexity analysis. Therefore, in section 3.3 we will assume that conceptually Daikon's space and time complexities are those of the simple incremental algorithm to provide for a simpler comparative analysis of our tool Turnip.

2.3. Other Tools Based on Dynamic Analysis

A lot of recent work has been done in the field of extracting models or specifications using dynamic techniques. Henkel and Diwan developed a tool that discovers high-level algebraic specifications from Java classes in the form of axioms using dynamic analysis [14, 15]. Whaley, Martin and Lam propose multiple finite state machine submodels to model the interface of a class with the purpose of formally specifying method call sequences. They use a combination of static and dynamic techniques to automatically extract such models [27]. Caffeine [12] is a tool for dynamic analysis of Java programs, which allows the developer to check a conjecture about the behavior of a Java program. For instance, the developer might inquire about the number of times a particular method was called during program execution. The tool works by generating a data trace during program execution and running a Prolog engine to perform queries over the trace. Caffeine was developed by Guéhéneuc, Douence and Jussien.

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.

Other research concentrates on aiding in software error detection. Hangal and Lam created DIDUCE - a tool which dynamically formulates constraints for a program and can inform the user when the formulated constraints are violated at runtime [13]. An automatic debugging tool called Carrot has been developed by Pytlik, Renieris, Krishnamurthi, and Reiss [24]. Carrot is based on Daikon and works by hypothesizing constraints that hold for a program by examining a (large) number of correct runs of the target program. Carrot then examines the faulty run and finds the locations in the program where the hypothesized constraints were broken. Liblit, Aiken, Zheng and Jordan have developed a low-overhead general sampling infrastructure for gathering information from executions from multiple users [18]. They have demonstrated how to use the data traces to discover software bugs.

We are looking into constraints specified in terms of program variables to help human developers understand the program's behavior and implementation. Daikon was developed with this goal in mind, and it represents the current state of the art in dynamic invariant detection. Hence, we chose Daikon as the most appropriate tool for inferring polymorphic constraints.

3. Dynamic Detection of Polymorphic Behavior

In this section we present our approach to inferring polymorphic behavior from data traces and its limitations. We call our version of Daikon augmented to consider runtime-refined cases Turnip, in accordance with the naming scheme

devised by Daikon developers[4].

3.1. Our Approach

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. 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.

The next section describes the modifications we made to the Daikon system to examine the actual runtime classes of polymorphic variables in order to discover runtime-refined constraints.

3.2. Modifications to Daikon

We provide a detailed description of our modifications for Chicory. Daikon[5] was augmented in a similar way.

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[6] of the visible variables at each program point. Each node in the tree represents a variable in the target program and is responsible for retrieving the runtime value of this variable. Figure 4 presents an example of a variable tree constructed by Chicory for the entry point of the `checkOut` method in the `Library` class shown in figure 3. Nodes in the tree abstract the data types of the variables they contain.

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. A polymorphic variable is a variable declared as a user-defined interface or a user-defined class that has subclasses. 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 subclasses. In the `DisplayItem` example, a variable declared as `DisplayItem figure` is 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 referenced as its *base variable*.

The group variable's implementation is based on the state design pattern [11]. This allows it to alter its behavior at run-time as 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 also introduced a similar mechanism for reading in values for polymorphic variables into Daikon's back-end. For a polymorphic variable, Daikon treats the variables which are fields of potential runtime classes other than the actual runtime class, as undefined. This approach allows us to reuse the statistical justification mechanism built into Daikon using the number of non-missing samples observed 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).

[4] Daikon is an Asian radish.

[5] In this section we are going to distinguish between Daikon's front end for Java, Chicory, and Daikon as a back end constraint inference engine. The rest of the paper refers to both as Daikon.

[6] 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.

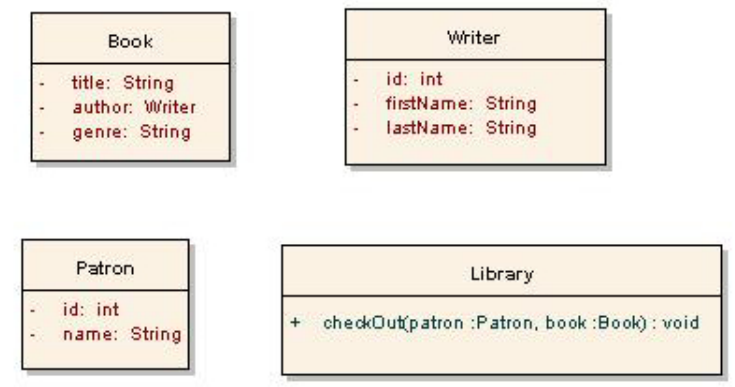

Figure 3. Classes for a library system.

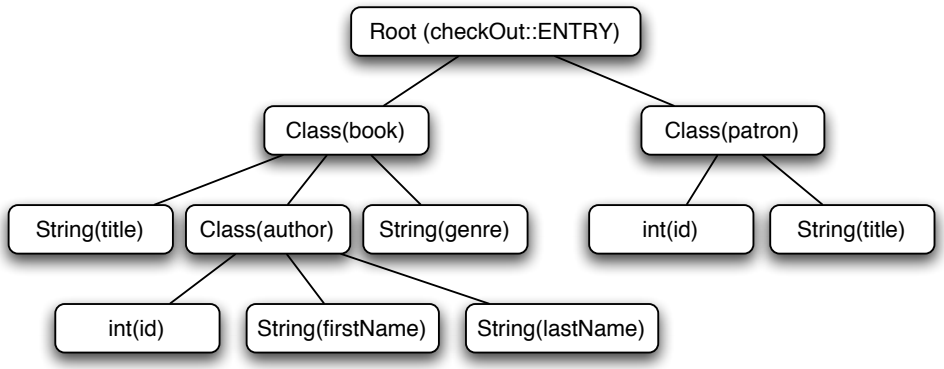

Figure 4. A (conceptual representation of a) variable tree constructed by Chicory for the entry point to `Library.checkOut` method.

3.3. Comparative Analysis of Daikon and Turnip

In this section we first analyze the space and time complexities of Turnip compared to Daikon. Next we argue that our modifications do not result in the loss of constraints given the same number of samples per program point.

By considering different runtime cases for polymorphic variables we increased the number of possible constraints, C, that Turnip creates at a program point with polymorphic variables. Following the naming conventions established in figure 2, the number of potential constraints that Daikon considers at program point P_1 is $C_D = G(V_D)$, where V_D is the total number of variables at P_1. Suppose, in the worst case that all variables V_D are polymorphic and each variable belongs to a different inheritance hierarchy $1..k$, where $k = V_D$. Let $l_1..l_k$ be the number of classes in the inheritance hierarchies $1..k$ respectively, such that $n_1..n_k$ is the maximum number of fields for classes in the respective hierarchy. Let V_T be the number of variables that Turnip considers at P_1. $V_T \leq n_1 * l_1 + ... + n_k * l_k = n_1 * l_1 + ... + n_{V_D} * l_{V_D} \leq n' * l' * V_D$, where $n' = max(n_1, ..., n_k)$ and $l' = max(l_1, ..., l_k)$.

Our modifications result in a linear increase on the number of variables considered at a program point with polymorphic variables. In the worst-case, 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 for a program point with polymorphic variables $G(V_T)$ is cubically larger than $G(V_D)$ in the size of the largest inheritance hierarchy and the maximum number of fields in the involved hierarchies. Total constraint detection time is linear in the number of potential constraints at a program point. 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 Daikon community suggests, most of

Figure 5. Hierarchy of subclasses for `A`.

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 [9].

Another possible 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. The details of this argument are provided in the following paragraphs.

First, consider Daikon's statistical justification mechanism. The probability that a relationship has occurred by chance is computed based on the number of samples observed on which this relationship held. For example, for a simple property "x is even", which has a 50 percent chance of having occurred by chance, the probability that "x is even" occurred by chance alone will be computed as $1 - (\frac{1}{2})^n$, where n is the number of samples of variable "x" encountered. If the probability above is less than the specified threshold, say 0.1, then "x is even" will be reported. For more complicated properties, which do not very likely occur by chance, such as "set X is an intersection of set Y and set Z", the probability that it occurred by chance may be set to below the threshold after a fixed number of samples observed. In other words, for more complicated properties only a few samples need to be observed to establish statistical significanse.

Second, consider the samples that Turnip observes for polymorphic variables. At a program point with a polymorphic variable Turnip will observe the same number of samples with the declared type of the polymorphic variable as Daikon. Consider the hierarchy in figure 5. If the variable `A a` is in scope at the program point P_1 and P_1 gets 10 samples, both Turnip and Daikon will observe the fields of `A a` 10 times and report the same constraints on the fields of `a`. However, the runtime type of `a` may be A_{11} 6 times and A_{12} 4 times out of the observed 10 samples. While Daikon does not consider the runtime type of `a`, Turnip examines the fields of the class A_{11} in 6 samples and the fields of the class A_{12} in 4 samples. The constraints on the fields of the class A_{12} may not be statistically justified due to the small number of samples observed. Turnip will have to observe more samples with A_{12} as the runtime type of `a` to justify the constraints on the fields of A_{12} statistically. However, the number of samples Turnip observes for the declared type of a polymorphic variable is always equal to the number of samples observed by Daikon. Turnip reports all the constraints Daikon does for the declared type of a polymorphic variable and, possibly, some polymorphic constraints on the runtime cases for the polymorphic variable if the number of samples observed for a runtime class justifies a constraint statistically.

3.4. 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 3.3. 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.

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.

4. Extended Examples

We present two real world examples to demonstrate runtime constraints produced by Turnip. Both cases offer an insight into the behavior of the system that Daikon alone could not.

4.1. Example Set Up

Test suite selection is important when dynamically inferring useful constraints [8, 9] because test cases determine what values are taken by variables in the target program. If a variable takes on too few values during the examined program runs it results in a small number of essential constraints inferred due to low statistical justification and larger number of accidental properties reported due to the lack of counter-examples. For example, let `int x` be a variable at a program point. If the only values for `x` Daikon observes are `10` and `15`, Daikon may output the constraint `x one of {10, 15}`. This constraint is the result of the two values observed, while the true constraint for `x` may be `x > 0`. To avoid over-specialization without placing too large a burden on the programmer to develop a comprehensive test suite, we wrote the tests so that they exercise each program point several times with randomly chosen values.

4.2. Money Example

This section reports actual relevant constraints detected by Daikon and Turnip for several methods declared in the `IMoney` interface of the `Money` example included with JUnit.

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 6.

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. Such a representation is used to defer exchange rate conversions, and to compute the total value of all monies in a `MoneyBag` in a single currency on the fly with the current exchange rates.

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 7 and 8.

Before we turn to the results in figures 7 and 8, let us consider the `IMoney add(IMoney m)` method 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 Money.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.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 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 )
```

A few comments about the notation in the above code fragment and figures 7 and 8 are in order. `A.b(X x):::EXIT` is an exit point for method `b` of class `A`, which takes a parameter `x` of declared type `X`. An exit point is followed by a list of postconditions. Symbol '&' denotes logical AND, symbol '==' stands for equality, and symbol '==>' (as well as '⇒') is logical implication.

All the runtime classes reported by Turnip for the corresponding variables have been 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.

In the `add` method of the `Money` class, for example, the case when the runtime class of `m` is a `MoneyBag` and the returned object is a `Money` may seem confusing. How can

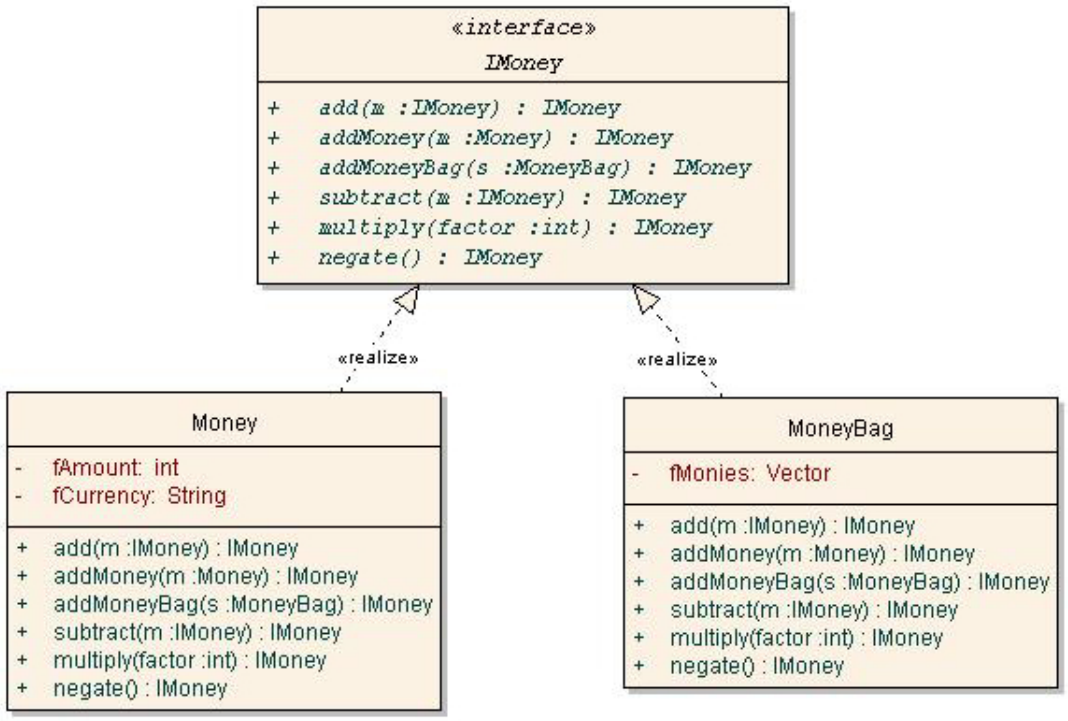

Figure 6. `Money` **Example Class Diagram**

we add a `MoneyBag` to a `Money` and get a `Money` back? Suppose we have a 12 CHF worth of `Money` to which we add a `MoneyBag` which contains -12 CHF and 3 USD. The result is a `MoneyBag` which contains 3 USD. However, a `MoneyBag` with only one currency is simplified to an equivalent `Money` object, which is returned from the `add` operation in this case.

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[7] to `null`.

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

Figures 7 and 8 present constraints that characterize the behavior of the following methods from the `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 the given `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 in our experiments Daikon did not infer the specified constraints 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.,

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 7. 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.

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 8. 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[]`.

`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.

4.3. Database Query Engine Model

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, such as "SHOW Close of IBM WHEN Date is 12/15/2005".

Our tests exercised some queries from the original system expressible in the language of our model. Our test queries include such queries as "SHOW (Low of IBM + High of IBM) * 0.5", "SHOW Return of IBM" and "SHOW Close of IBM + Return of IBM".

The query engine creates an executable query based on a parsed request. An executable query is a tree of nodes, where each node is capable of returning its value for a particular date and time. In our model the time series is presented as an array of doubles and a date is represented as an index into this array.

We extracted eight classes from the original query engine. Our model supports addition (`ExecAddOper`), multiplication (`ExecMultOper`) and negation (`ExecMinusOper`) of entities, which can be of two types: a constant (`ExecConstant`) and a relation column (`ExecRelationColumn`). A relation column represents a column, such as Close, of a relation, such as IBM. The relationships among the classes are presented in figure 9.

The `getValue(int idx)` method in the abstract class `ExecNode` is intended 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. For example, the `ExecRelationColumn` returns the value at `idx` of the column of its relation, while the `ExecAddOper` returns the sum of the value of its left node `left` and its right node `right`.

To reflect the logic of the `getValue(int idx)` method we introduced a new variable into Daikon and Turnip, which is the element at a particular position in an array, e.g. `vals[idx]`, where `vals` is an array, and `idx` is an integer. The position is provided by a suitable integer variable visible at the same program point as the array.

The constraints characterizing the behavior of the query engine model inferred by Daikon and Turnip are summarized in figure 10.

No polymorphic variables were used in the base query classes, `ExecConstant` and `ExecRelationColumn`. Both Daikon and Turnip had no trouble detecting that the `getValue` method in class `ExecConstant` returns the constant's value and the same method in class `ExecRelationColumn` returns the element at index `idx` in its time series array.

Our test cases reflected only commonly used queries

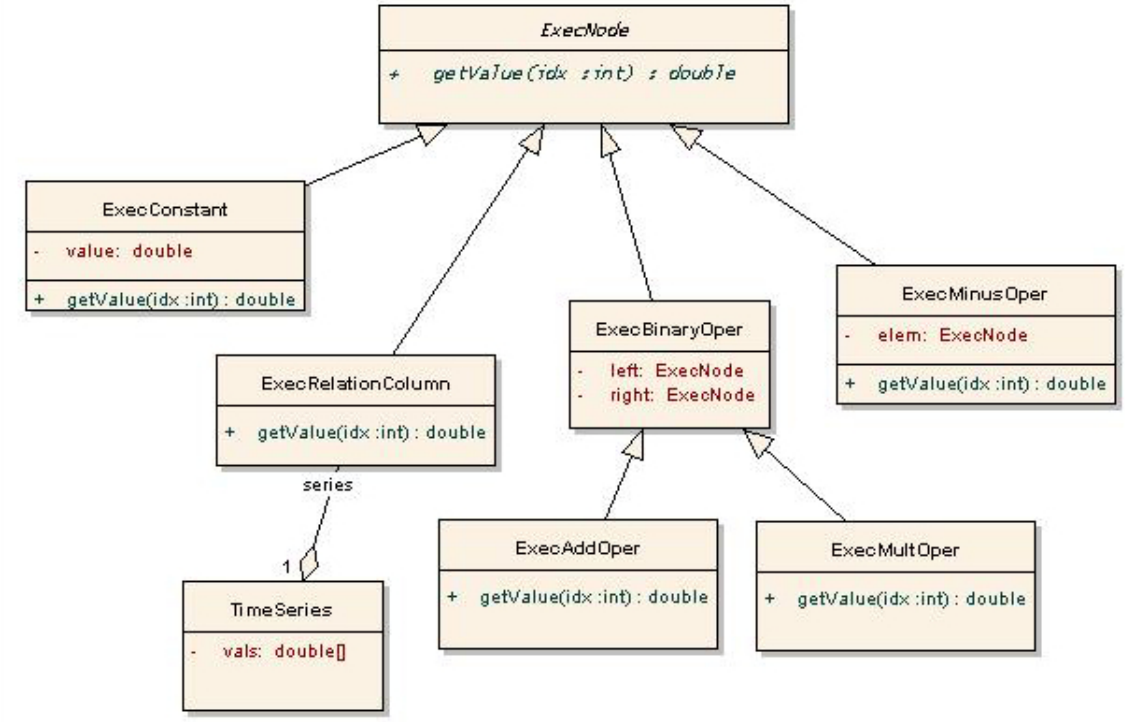

Figure 9. Database Query Engine Model Class Diagram

Class	*Postcondition*	*Daikon*	*Turnip*
ExecConstant	return == this.value	√	√
ExecRelationColumn	return == this.series.vals[idx]	√	√
ExecAddOper	(this.right.class == ExecRelationColumn & this.left.class == ExecRelationColumn) ⇒ (return == (this.right.series.vals[idx] + this.left.series.vals[idx]))	–	√
ExecMultOper	(this.right.class == ExecRelationColumn & this.left.class == ExecConstant) ⇒ (return == this.right.series.vals[idx] * this.left.value)	–	√
ExecMinusOper	(this.elem.class == ExecRelationColumn) ⇒ (return + this.elem.series.vals[idx] == 0)	–	√

Figure 10. Postconditions for the `getValue(int idx)` method overriden by classes in the `ExecNode` hierarchy.

which did not cover all potential runtime classes for the `left` and `right` nodes of the `ExecAddOper` and `ExecMultOper` classes and the `elem` node of the `ExecMinusOper` class. Therefore, Turnip only inferred constraints for the runtime classes it encountered in the examined program runs. This is shown in figure 10.

Turnip inferred constraints characterizing the behavior of the `getValue(int idx)` method in the `ExecAddOper`, `ExecMultOper` and `ExecMinusOper` classes for the runtime cases it has examined. Daikon was not able to discover any of the constraints for the `ExecAddOper`, `ExecMultOper` or `ExecMinusOper` classes listed in figure 10 because it examined only the declared types of their members which in each case is `ExecNode`, an abstract class that does not specify the behavior of its children.

5. 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. 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 would also like to explore the feasibility of dynamically detecting OCL-specified constraints [26] in object-oriented systems.

The big picture of our future work involves automatically discovering OCL constraints from code with the purpose of presenting them on design models to specify valid behavior. Designers or developers can then work with constraints (add, modify, delete them) without the extra burden of having to specify all of them by hand. OCL is a declarative language [26], which allows the constraints from the model to be easily evaluated, for instance, as an `assert` statement, in the corresponding code to make sure that the specified restriction indeed holds. This is an ongoing effort that is part of the Wyoming Programmers' Workbench project.

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. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[9] 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.

[10] 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.

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

[12] Y. Guéhéneuc, R. Douence, and N. Jussien. No Java without caffeine – a tool for dynamic analysis of Java programs. In W. Emmerich and D. Wile, editors, 17^{th} *conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.

[13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM Press.

[14] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[15] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 449–458, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, 2001.

[17] 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.

[18] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[19] T. Ne Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 2003.

[20] T. Ne Win, M. D. Ernst, S. J. Garland, D. Kırlı, and N. Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.

[21] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[22] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[23] 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.

[24] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, Ghent, Belgium, September 8–10, 2003.

[25] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 302–312, Orlando, Florida, May 22–24, 2002.

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

[27] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, 2002.

[28] T. Xie and D. Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.