Class-Invariant Binary Relations in Java

Adam Brooks Webber
University of Wisconsin - Milwaukee
webber@cs.uwm.edu

July 12, 1999

Abstract

In this paper we introduce a new method for automatically identifying class invariants in object-oriented programs. We define a category of class-invariant relations for Java programs and give a static analysis technique that finds binary relations in this category. Our experimental system, Bounder, demonstrates one application of this analysis technique: the identification of unnecessary array bounds checks in Java programs.

1 Introduction

In the Java programming language there is a predefined class named `Vector` which is used to collect lists of objects, especially when the programmer cannot tell in advance how large the list will be. A `Vector` object provides these methods (among others):

`addElement(x)`: add a new object `x` to the end of the list
`elementAt(i)`: return the `i`'th object from the list
`indexOf(x,i)`: return first index of `x` after `i` in the list, or `-1` if none

A `Vector` object does its job using two private pieces of data:

`elementData`: an array of the objects in the list
`elementCount`: the number of objects in the list

The array stored in `elementData` contains the list in question. When a `Vector` object is created, `elementData` is initialized to an array with room for some small number of objects, and `elementCount` is initialized to zero. As objects are added, the `elementData` array fills up and the `elementCount` counter grows. When an object is added to a `Vector` that already has a full `elementData` array, a new and larger array is allocated, and the old contents are copied into the new one.

Although `elementData` and `elementCount` are changed by the `addElement` method, there is a binary relation on them that does not change: it is always true that

$$\texttt{elementCount} \leq \texttt{elementData.length}$$

This is an example of a class invariant: an assertion that is always true about objects of a given class.[1]

Class invariants are ubiquitous in object-oriented programming. Programmers almost always have invariants in mind when designing a class, although they are rarely documented and almost never proved correct. Not all class invariants are binary relations, of course. But our research focuses on those that are. To see why such relations are useful, consider the code for the `indexOf` method of the `Vector` class [BP97].

[1] Whether the assertion is always true depends on what the definition of *always true* is. We will provide one.

```
1   public final synchronized int indexOf(Object elem, int index) {
2     for (int i = index ; i < elementCount ; i++) {
3       if (elem.equals(elementData[i])) {
4         return i;
5       }
6     }
7     return -1;
8   }
```

The Java virtual machine performs array bounds checks on all array references. Such bounds checks must also be performed by native code generated for Java programs unless the compiler can eliminate them using static analysis. The standard compiler technique for optimizing such bounds checks is as follows. The compiler identifies loops with iteration variables, and looks for obvious bounds on the iteration variable. For example, from line 2 above, and since `i` is not modified in the loop body, it is clear that `i` is bounded between `index` and `elementCount`. Bounds on the iteration variable can be generalized to give bounds on induction variables in the loop. These bounds can be used to justify the elimination of unnecessary bounds checks or, more generally, to justify the movement of upper and/or lower bounds checks out of the loop. [Muc97]

In the code above, such analysis cannot eliminate all the unnecessary bounds checks from line 3. It can move the lower bounds check into the loop header; an aggressive compiler might even change the upper bound of the loop to min(`elementCount`, `elementData.length`), remove the upper bounds check from the loop body, and insert a corresponding check after the loop. But it is a class invariant that `elementCount` $\leq$ `elementData.length`, as we have already noted. So min(`elementCount`, `elementData.length`) = `elementCount` and no check on the upper bound is required.

This example shows one way in which class-invariant binary relations can be used to support a standard compiler optimization. Others immediately suggest themselves. Consider again the operation on line 3. It actually performs three unnecessary checks: the lower bounds check, the upper bounds check, and also a null pointer check for the `value` field. But `value` $\neq$ `null` is another class-invariant binary relation, so that check too could be eliminated.

In fact, binary relations are an extraordinarily useful form of information. The (binary) relation of equality between two variables, or between a variable and a constant, is the key precondition for a wide variety of optimizations. One class of these optimizations, the class of *thinnings*, was studied in [Web95]: it includes common subexpression elimination, loop jamming, loop folding, dead code elimination, code sinking, loop invariant removal, and other, more exotic transformations. Loop guards and guards on conditionals are commonly expressed as tests of binary relations like $=$, $\neq$, and $<$; so being able to say something about the binary relations between variables allows a compiler to reason about loop termination and branch direction.

This paper introduces a new approach that makes it possible to identify a category of class-invariant binary relations automatically. After a review of related work in Section 2, we start in Section 3 with a definition of a category of invariants, the *DC invariants*, that is reasonably tractable for static analysis. In Section 4 we describe a static analysis that identifies DC-invariant binary relations. Section 5 presents a detailed application of the analysis as performed by our experimental system. We conclude in Section 6 with a list of some interesting directions for further research on class invariants.

2 Related Work

There is little prior work on this problem, although the concept of class invariant is well known in the literature.

The term *class invariant* appears in the software engineering literature as a concept of object-oriented analysis and design. The terms *constraint* and *assertion* are used along with *class invariant* to refer various kinds of restrictions on the states that may be assumed by an object

[MP92, Ner92]. This use is only glancingly related to the present work, but it is interesting evidence that class invariants are routinely designed into object-oriented programs.

The language Eiffel [Mey91] has syntax to support the direct statement by the programmer of class invariants as well as preconditions, postconditions, and other constraints. These may be compiled into runtime checks but are not checked statically. ANNA (ANNotated Ada) [Luc91] is another example of a language that supports explicit constraints that may be compiled into runtime checks.

Extended static checking, including the checking of class invariants, is the focus of the ESC project [DLNS98]. In this project, the power of a theorem-proving engine is brought to bear on the problem of verifying the programmer-supplied invariants. Several of the same problems we encountered were pointed out by ESC project researchers [LS97]: in particular, the difficulties caused by constructors that can make incompletely-constructed objects visible to other parts of the program, and the difficulties caused by the use of wide scopes in which fields can be altered.

Another project that applies heavyweight theorem proving to the problem of verifying program properties is the LOOP project [JvdBH+98]. The LOOP tool translates Java classes into a higher order logic, and then applies a back-end theorem prover (the PVS system) to prove various properties of the program, including class invariants. The project goals include the definition of a language extension, "Annotated Java", that will allow the explicit statement of correctness assertions.

In all the examples above, class invariant relations are supplied by the programmer. Some projects attempt to verify these invariants using automatic tools, but none propose to identify invariants automatically.

3 A Category of Class Invariants for Java

In this section we define a category of class invariants for Java. Many such definitions are possible, and we suggest several variations. But the definition we focus on here is one that is proving to be both useful and amenable to static analysis. We begin by distinguishing two kinds of nonstatic fields, two kinds of constructors, and two kinds of nonstatic methods.

Definition 1 *A* disciplined *nonstatic field of a class is one that is written only in constructors or in synchronized nonstatic methods of the class, and only as a field of* `this` *(the self object). All other nonstatic fields are* undisciplined.

Definition 2 *A* tight *constructor is one that does not store a reference to the object under construction (except on the operand stack), either directly or by passing it to a method which stores it. All other constructors are* leaky.

Definition 3 *A* clean-called *nonstatic method one that is not called, either directly or transitively, from any constructor of the class, or from any nonstatic method of the class at any point at which any disciplined nonstatic field has been altered. It is either explicitly synchronized, or else called only from other clean-called nonstatic methods of the class. All other nonstatic methods are* dirty-called.

Definition 4 *A nonstatic* DC invariant *(disciplined clean invariant) is a relation involving only constants and/or the disciplined nonstatic fields of a class, that is always true on entry in clean-called nonstatic methods of the class.*

Many other definitions of class invariants are possible, of course. More specific definitions are tempting, since an invariant that is true in more places will be more generally useful. But suppose, for example, that we restrict to invariants that are true at all points in all clean-called methods, not just on entry. Then we would miss the (we believe important) category of invariants that are broken temporarily in a method, but restored before the method exits. Or suppose we restrict to invariants that are true on entry in all methods, not just in clean-called methods. Then we would miss another common programming pattern: the invariant that is temporarily

broken but then reestablished by a method call. Such an invariant does not hold on entry in the method that is called to reestablish it (a dirty-called method), but it may still hold on entry in all clean-called methods.

More general definitions are also tempting. For example, it is tempting to eliminate the restriction to disciplined fields. But more general definitions may be less tractable computationally. The following theorem shows why DC invariants are reasonably amenable to static analysis.

Theorem 1 *The disciplined nonstatic fields of* `this` *on entry in any clean-called nonstatic method of the class must be as they were on exit from some constructor or nonstatic method of the class, excluding the case of abrupt exit from a tight constructor.*

Proof: consider any time in any execution when any clean-called nonstatic method m is entered. By definition, no parent in the call stack can have made any changes to the disciplined nonstatic fields, nor can a parent in the call stack be the constructor that is initializing the object. Since m and all methods that alter disciplined nonstatic fields are synchronized on the same lock, we also know that no other method invocation in another thread can be changing the disciplined nonstatic fields while m executes. Therefore the disciplined nonstatic fields must be as they were on abrupt or normal termination of some previous invocation of a method or constructor of the class. Further, we can eliminate the case of abrupt exit from a tight constructor since, if a tight constructor terminates abruptly, the object reference being constructed is lost, and so could not be used as `this` in any subsequent call.

Actually, Definitions 1 through 3 may be relaxed somewhat: they are sufficient but not necessary conditions for Theorem 1. For example, we might require synchronized regions of code (instead of entire synchronized methods) in the definitions of disciplined fields and clean-called methods. Or tight constructors might be permitted to store the reference to the object under construction, as long as they cannot terminate abruptly before all writes to the disciplined fields have taken place. For analysis of a program which is single-threaded, the synchronization requirements can be completely eliminated. We are experimenting with these and other variations.

In the preceding definitions we have carefully specified only nonstatic fields and nonstatic methods. Exactly parallel definitions can be given for disciplined static fields, clean-called static methods, and static DC invariants, substituting class initialization code for constructors in the definitions. We can then give a parallel invariance theorem for static fields and methods:

Theorem 2 *The disciplined static fields of a class on entry in any clean-called static method of the class must be as they were on exit from class initialization code or from some static method of the class.*

Unfortunately, we cannot combine static and nonstatic DC invariants into a single category, since static methods and nonstatic methods synchronize on different locks. This means in particular that our DC invariants do not include relations involving static fields that are invariant on entry in nonstatic methods. (For single-threaded programs the two definitions could be combined.)

To decide whether a field is disciplined, a constructor is tight, or a method is clean-called, we clearly need access to the whole program in question, not just the individual class in isolation. Modifying the definitions to allow the analysis of individual classes in isolation, without seeing the rest of the program, is possible. There are several problems to be solved. There is the problem that unseen parts of the program may modify the fields involved in invariants; we could solve this by requiring that disciplined fields be declared `private`. There is the problem that unseen parts of the program may be called by methods of the class at times when an invariant does not hold, and may in turn call methods of the class that expect the invariant to be true; we could try to solve this by requiring that no method in the class can call outside the class after altering any disciplined fields, but this leaves a loophole involving method overriding. A method of the class may alter disciplined fields and then call another (dirty-called) method of the class. If that dirty-called method is overridden in a subclass, it could call a (formerly clean-called) method. We could solve this by requiring that dirty-called methods be declared `private` or `final`.

But this is an exercise we have not pursued further. Clearly, with enough restrictions, we can create some category of invariants for individual classes in isolation. But we suspect that any such definition will prove to be too strict—that an analyzer based on it will find too few invariants in practice, since programmers do not always specify `private`, `synchronized` and/or `final`, even where the class design allows them. (Finding invariants for a class in isolation is the kind of problem that makes us wish for richer sources, like those with the programmer-supplied annotations used in the ESC project [DLNS98].) In any event, for the application we have in mind—analysis to support optimization of native code generated from Java—the whole-program assumption is a reasonable one.

4 Analysis of Nonstatic DC Invariants

Here we discuss an analysis for finding nonstatic DC-invariant binary relations. (The analysis for static DC-invariant binary relations is similar.) This analysis requires relational analysis of individual methods, an intra-class analysis that extracts DC-invariants from the individual method results, and a whole-program analysis to perform the classifications of Definitions 1 through 3. We explain the method analysis and the intra-class analysis below. The whole-program analysis is conventional and will not be discussed further.

4.1 Individual Method Analysis

Our goal at this level is to extract information about binary relations on the input and output values of an individual nonstatic method or constructor. For each nonstatic method or constructor m in class C, we define a set of canonical variables whose relations are of special interest:

$$\begin{aligned}
\text{Const} &= \{\texttt{0}, \texttt{1}, \texttt{null}, \texttt{false}\}\text{, the canonical constants} \\
\text{IField}_C &= \text{the set of nonstatic disciplined fields of class } C \\
\text{InIField}_m &= \{x_{\text{in}} | x \in \text{IField}_{\text{classof}(m)}\}\text{: field values on entry in } m \\
\text{OutIField}_m &= \{x_{\text{out}} | x \in \text{IField}_{\text{classof}(m)}\}\text{: field values on exit in } m \\
\text{In}_m &= \{x | x \text{ is a formal parameter of method or constructor } m\} \\
\text{Out}_m &= \{m\text{'s returned value}\}\text{, or } \emptyset \text{ if none} \\
\text{Canon}_m &= \text{Const} \cup \text{InIField}_m \cup \text{OutIField}_m \cup \text{In}_m \cup \text{Out}_m
\end{aligned}$$

The canon for a method is the set of variables on which we want to find relations: relations that hold for every possible execution of the method in the given program. The canon for a method includes a set Const of suggested canonical constants. The actual constants used here would depend on the application. For eliminating unnecessary array bounds checks, for example, we want to know the relation between the length of an array and the constant 0; so the constant 0 should be part of the canon for every method.

Information about binary relations on canonical variables can be represented as a *canon table*, a table of relations indexed by pairs of canonical variables. We can then view any relational method analysis as a function that takes a canon table reflecting any assumptions—for example, known entry conditions—and produces a canon table reflecting the conclusions that were reached from there.

$$\begin{aligned}
\text{Rel} &= \text{the vocabulary of expressible relations} \\
\text{CanonTable}_m &= \text{Canon}_m \times \text{Canon}_m \to \text{Rel} \\
J_m : \text{CanonTable}_m \to \text{CanonTable}_m &= \text{the method analysis function}
\end{aligned}$$

The initial analysis of methods can be made using no assumptions, while the initial analysis of constructors can be made using slightly stronger assumptions reflecting Java's default initializa-

tion of fields:

$$\begin{aligned} \top_m \in \text{CanonTable}_m &= \text{the table of } \top \text{ relations} \\ \text{default}_m \in \text{CanonTable}_m &= \text{the table using default initializers} \end{aligned}$$

For the intra-class analysis below, we will need separate analyses for normal termination and abrupt termination. So we need to define three method analysis functions:

$$\begin{aligned} A_m : \text{CanonTable}_m \to \text{CanonTable}_m &= \text{abrupt exit analysis} \\ N_m : \text{CanonTable}_m \to \text{CanonTable}_m &= \text{normal exit analysis} \\ J_m &= A_m \sqcup N_m \end{aligned}$$

In our prototype, the method analysis functions are implemented using the relational constraint technique. The technique is described in [Web97], and the earlier project for which the technique was developed is described in [Web95, Web93]. Relational constraint constructs a table of binary relations and treats the program as a collection of constraints on tuples of relations in the table. The resulting table gives binary relations not only on the canonical variables but on all variables in the SSA form [CFR+89] of the method being analyzed. The desired canon tables are subtables of this full table of relations. Use of this technique makes the analysis functions monotonic: strengthening any relation in the table of initial assumptions cannot result in the weakening of any relation in the table of conclusions.

Using the relational constraint method, we can compute the analysis functions in $O(n^2)$ time, where n is the number of variables in the SSA form of the method being analyzed. Since this is an implementation of earlier work, we will omit further discussion, and concentrate instead on the problem of combining the individual method analyses to extract DC invariants.

4.2 Intra-Class Analysis for Nonstatic DC Invariants

In this section we consider the problem of how to extract nonstatic DC invariants from the individual method analysis functions.

For the class as a whole we will define a restricted type of canon table:

$$\begin{aligned} \text{Canon}_C &= \text{Const} \cup \text{IField}_C \\ \text{CanonTable}_C &= \text{Canon}_C \times \text{Canon}_C \to \text{Rel} \end{aligned}$$

We also define a pair of functions, g_m and g_m^{-1}, for converting between class canon tables and method canon tables. Function g_m takes a class canon table and produces a method canon table by mapping each field in the class canon ($x \in \text{IField}_C$) to the corresponding field *on entry* in the method canon ($x_{\text{in}} \in \text{InIField}_m$), and supplying $\top$ relations for those pairs not included in the class canon table. Function g_m^{-1} works in the opposite direction, taking a method canon table and producing a class canon table by mapping each field *on exit* in the method canon ($x_{\text{out}} \in \text{OutIField}_m$) to the corresponding field in the class canon ($x \in \text{IField}_C$).

$$\begin{aligned} g_m &: \text{Canon}_{\text{classof}(m)} \to \text{Canon}_m \\ g_m^{-1} &: \text{Canon}_m \to \text{Canon}_{\text{classof}(m)} \end{aligned}$$

Now by Theorem 1, nonstatic DC invariants for the class will be a class canon table Q such that

$$A = \bigsqcup_{\text{tight constructors } x} g_x^{-1}(N_x(\text{default}_x)) \tag{1}$$

$$B = \bigsqcup_{\text{leaky constructors } x} g_x^{-1}(J_x(\text{default}_x)) \tag{2}$$

$$C = \bigsqcup_{\text{nonstatic dirty-called methods } x} g_x^{-1}(J_x(\top_x)) \tag{3}$$

$$Q = A \sqcup B \sqcup C \sqcup \bigsqcup_{\text{nonstatic clean-called methods } x} g_x^{-1}(J_x(g_x(Q))) \tag{4}$$

We can find a solution to Equation 4 by iteration:

$$K(Z) = Z \sqcup \bigsqcup_{\text{nonstatic clean-called methods } x} g_x^{-1}(J_x(g_x(Z))) \quad (5)$$

$$Q_0 = A \sqcup B \sqcup C \quad (6)$$

$$Q_{i+1} = K(Q_i) \quad (7)$$

Because the method analysis functions (implemented by relational constraint) are monotonic, the iteration function K is monotonic: $Y \sqsubseteq Z \Rightarrow K(Y) \sqsubseteq K(Z)$. Any solution Q to Equation 4 must have $Q_0 \sqsubseteq Q$, and using the monotonicity of K we have that for all i, $Q_i \sqsubseteq Q$. We can conclude that the iteration of Equation 7 converges to a least fixpoint, which in this context is a strongest class-invariant canon table.

How many iterations are required? In the worst case, each Q_{i+1} is a table that differs from Q_i in only one entry. A given entry can change only a constant number of times (since the relational constraint vocabulary is a fixed-height lattice) but even so we might have to iterate $O(|\text{Canon}_C|^2)$ times to reach a fixpoint. Unfortunately these iterations are expensive, since each one recomputes the relational analysis of every clean-called method in the class.

4.3 A Faster Approximation

It is possible to safely approximate the functions J_x using information available in $J_x(\top_x)$. In this way we will be able to perform the iteration above without having to compute each method analysis function more than once.

The function g_x^{-1} throws away many of the relations from the method canon table, truncating it to a class canon table. Some of that information is potentially quite useful, since *it includes relations between the input and output values of the disciplined fields.* These relations can be used to approximate J_x, since any strengthening of input relations in this table may lead, using only transitivity within the table, to a strengthening of the output relations.

Consider for example this method:

```
1   void ex() {
2     if (this.x < this.y) this.x = this.y
3   }
```

Let S be the initial method canon table $S = J_{\mathtt{ex}}(\top_{\mathtt{ex}})$ and let R be the resulting class canon table $R = g_{\mathtt{ex}}^{-1}(S)$. $R(\mathtt{x}, 0) = (\top)$, showing that we end up with no information about the final relation between `x` and 0. But S contains information that was thrown away by $g_{\mathtt{ex}}^{-1}$: in particular, $S(\mathtt{x}_{\text{in}}, \mathtt{x}_{\text{out}}) = (\leq)$. Now suppose we are given a new class canon table U embodying stronger entry conditions, with $U(x, 0) = (>)$. We want the new method canon table $S' = J_{\mathtt{ex}}(g_{\mathtt{ex}}(U))$. We can compute a safe approximation $\hat{S}'$ by starting from $S \sqcap g_{\mathtt{ex}}(U)$ and then performing the strengthenings implied by transitivity. In this example, the final table $\hat{S}'$ will have $g_{\mathtt{ex}}^{-1}(\hat{S}')(\mathtt{x}, 0) = (>)$.

In general, let $\hat{J}_x$ be the approximation to J_x computed from $J_x(\top_x)$ as described above. This is a conservative monotonic approximation, and can be used in place of J_x in the iteration of Equation 7. It remains to be seen how useful this approximation will be in practice, but it is interesting to note that it is adequate for the example in the next section.

5 Example

We have implemented a DC invariant analyzer as part of an experimental system called Bounder. This section presents a detailed example of an analysis performed by Bounder.

Bounder tries to identify unnecessary array bounds checks in Java programs. It works from Java bytecode, decompiling each method into an SSA control-flow graph. It applies a relational constraint analysis to each method, then extracts DC invariants as outlined above, using the approximate analysis functions $\hat{J}$ as just described.

Bounder uses the vocabulary of integer relations from [Web97]. To avoid burdening the reader with this vocabulary, we report the results of the analysis below in terms of simple integer relations: $<$, $=$, and so on. The actual relations derived by our prototype are somewhat stronger than those reported here.

For this example we constructed a class called `MiniVector`, using all of the field definitions, all of the constructor definitions, and three of the method definitions from the `Vector` class of Java.

```
1   public class MiniVector {
2     protected Object elementData[];
3     protected int elementCount;
4     protected int capacityIncrement;
5     public Vector() {
6       this(10);
7     }
8     public Vector(int initialCapacity) {
9       this(initialCapacity, 0);
10    }
11    public Vector(int initialCapacity, int capacityIncrement) {
12      super();
13      this.elementData = new Object[initialCapacity];
14      this.capacityIncrement = capacityIncrement;
15    }
16    public final synchronized void
17      addElement(Object obj) { ... }
18    private void
19      ensureCapacityHelper(int minCapacity) {...}
20    public final synchronized int
21      indexOf(Object elem, int index) {...}
22  }
```

The bodies of the methods are shown below, where each method analysis is discussed. For this example we also constructed a dummy main method that creates a `MiniVector` and calls all its methods. This is necessary since the analysis operates on entire programs, not individual classes in isolation. All the fields are disciplined, all the constructors are tight, and all the methods are clean-called.

For each of the three constructors x, the canon table $N_x(\text{default}_x)$ gives the relations on canonical objects for normal completion of that constructor, starting from the relations established by the default initializers. For the constructors at lines 8 and 11 we will have

$$N_x(\text{default}_x)(\texttt{elementData.length}_{\text{out}}, \texttt{elementCount}_{\text{out}}) = (\geq)$$

showing that on normal completion `elementData.length` is greater than or equal to `elementCount`. For those two constructors we also have

$$N_x(\text{default}_x)(\texttt{elementData.length}_{\text{out}}, \texttt{initialCapacity}) = (=)$$

showing that the initial capacity request has been obeyed. (Actually, the detailed relation shows that `elementData.length` and `initialCapacity` are equal *and non-negative*. If the initial capacity request had been negative the constructor would not have completed normally, so in the N_x tables only positive capacity requests are reflected.) This relation is used in the analysis of the no-arg constructor x at line 5, resulting in an even stronger relation between `elementData.length` and `elementCount`:

$$N_x(\text{default}_x)(\texttt{elementData.length}_{\text{out}}, \texttt{elementCount}_{\text{out}}) = (>)$$

So if the no-arg constructor is used and completes normally, `elementData.length` is known to be greater than `elementCount`. For all three constructors x the initial value of `elementCount` is

known to be zero:

$$N_x(\text{default}_x)(\texttt{elementCount}_{\text{out}}, 0) = (=)$$

Since there are no leaky constructors or dirty-called methods, the initial class canon table Q_0 is just the union from all the constructors. This has

$$Q_0(\texttt{elementData.length}, \texttt{elementCount}) = (\geq)$$

The `ensureCapacityHelper` method is the one that actually does the work of extending the `elementData` array as necessary.

```
1   private void ensureCapacityHelper(int minCapacity) {
2     int oldCapacity = elementData.length;
3     Object oldData[] = elementData;
4     int newCapacity = (capacityIncrement > 0) ?
5       (oldCapacity + capacityIncrement) : (oldCapacity * 2);
6     if (newCapacity < minCapacity) {
7       newCapacity = minCapacity;
8     }
9     elementData = new Object[newCapacity];
10    System.arraycopy(oldData, 0, elementData, 0, elementCount);
11  }
```

Analysis of this method without any information about the relations on entry does not produce any information about the final relation between `elementCount` and `elementData.length`. It does however discover that, in the case of normal completion, `elementData.length` $\geq$ `minCapacity`; and also that in either normal or abrupt completion, `elementCount` is unchanged and the new length of `elementData` is at least as great as the old length. That is, for $x =$ `ensureCapacityHelper`,

$$\begin{aligned} N_x(\top_x)(\texttt{elementData.length}_{\text{out}}, \texttt{minCapacity}) &= (\geq) \\ J_x(\top_x)(\texttt{elementCount}_{\text{in}}, \texttt{elementCount}_{\text{out}}) &= (=) \\ J_x(\top_x)(\texttt{elementData.length}_{\text{in}}, \texttt{elementData.length}_{\text{out}}) &= (\leq) \end{aligned}$$

The `addElement` method is the one that adds a new element to the end of the `MiniVector`.

```
1   public final synchronized void addElement(Object obj) {
2     int newcount = elementCount + 1;
3     if (newcount > elementData.length) {
4       ensureCapacityHelper(newcount);
5     }
6     elementData[elementCount++] = obj;
7   }
```

Using the relation between the parameter to `ensureCapacityHelper` and the resulting value of `elementData.length`, the analysis of `addElement` concludes that `elementData.length` > `elementCount` at the beginning of line 6. (So the upper bounds check there is seen to be unnecessary without the help of any class invariant.) It discovers that in the case of normal completion, `elementData.length` $\geq$ `elementCount`, while in the case of abrupt completion `elementCount` is unchanged; and that in either case the new `elementData.length` at least as great as the old one.

Although the methods do not (analyzed from $\top_x$) establish the invariant that `elementCount` $\leq$ `elementData.length`, they do maintain that invariant: they either establish it or leave `elementCount` unchanged and `elemenData.length` no less than it was. So the iteration of Equation 7 results in a fixpoint Q with $Q(\texttt{elementData.length}, \texttt{elementCount}) = (\geq)$.

Finally, re-analysis of the method `indexOf` (which was shown in the example at the beginning of this paper) using the invariant supports the elimination of the upper bounds check in the loop.

Interestingly, the analysis does not conclude that `elementCount` ≥ 0, even though the field is initialized to 0 and is changed only by being incremented. This is because of the possibility of

overflow at line 6 in `addElement`. In fact such overflow will not occur, since the array allocation in `ensureCapacityHelper` must fail long before. But we are not certain how to incorporate such reasoning soundly into the analysis. In any event we could not eliminate the lower bounds check from the loop in `indexOf` (at least not without knowing more about the point of call) since the iteration starts from a value passed in as a parameter.

6 Conclusion

In this paper we have presented one category of class invariants (the DC invariants), one static analysis for finding them (using Theorem 1), one application of them (the identification of unnecessary array bounds checks in Java programs), and one example (`MiniVector`). We obviously cannot conclude from such evidence that class invariant analysis will be practical. Rather, we consider this to be evidence that class invariant analysis is worth further investigation.

Bounder works on a selection of toy examples like the one presented above, but it does not yet handle the full Java language. When development of Bounder is complete, we plan to use it to analyze a variety of large Java systems (including itself). It will be interesting to see whether it finds a significant number of unnecessary array bounds checks not identified by other methods, how often the $\hat{J}$ approximation is adequate, and what its performance is like on large programs.

We conclude by pointing out some of the many other avenues for further work on class invariant analysis.

- Developing better approximations for the method analysis functions. We approximate J_x by using transitivity constraints on the table $J_x(\top_x)$. But surely better approximations are possible. J_x could itself be represented as some kind of constraint. Is there a way to automatically collapse the constraint system used in the analysis of method x into a single large (conservatively approximate) constraint for J_x?

- Putting the class-invariance theorem on a solid semantic foundation. There are several ongoing projects to develop formal semantics for Java [Pus98, DEK99]. Is it possible to derive our Theorem 1 from a formal semantics?

- Identifying other categories of class invariants. Are there other definitions of class invariants that are tractable for static analysis, in Java or in other languages? Are there language constructs that would help?

- Exploring other applications for class invariants. For array bounds checks we needed integer relations. Can one define a vocabulary of binary relations on other types—object references, for example—that would be useful for other kinds of optimization?

References

[BP97] Lee Boynton and Jonathan Payne. Java class java.util.Vector, December 1997. Version 1.38, Sun Microsystems Inc.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[DEK99] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Compaq SRC Resarch Report 159, 1998.

[JvdBH+98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *OOPSLA '98, Object-Oriented Programming Systems, Languages and Applications*, October 1998.

[LS97] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Report 1997-007, Digital Equipment Corporation Systems Research Center, January 1997.

[Luc91] David Luckham. *Programming With Specifications: An Introduction to Anna, a Language for Specifying Ada Programs*. Springer Verlag, 1991.

[Mey91] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[MP92] David E. Monarchi and Gretchen I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, September 1992.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Ner92] Jean-Marc Nerson. Applying object-oriented analysis and design. *Communications of the ACM*, 35(9):63–74, September 1992.

[Pus98] Cornelia Pusch. Formalizing the Java virtual machine in Isabelle/HOL. Technical report, Technische Universität München, June 1998.

[Web93] Adam Brooks Webber. *Principled Optimization of Functional Programs*. PhD thesis, Cornell Univ., Ithaca, N.Y., August 1993.

[Web95] Adam Brooks Webber. Optimization of functional programs by grammar thinning. *ACM Transactions on Programming Languages and Systems*, 17(2):293–339, March 1995.

[Web97] Adam Brooks Webber. Program analysis using binary relations. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 249–260, June 1997.