

Automatic Detection of Immutable Fields in Java*

S. Porat[†] M. Biberstein[†] L. Koved[‡] B. Mendelson[†]

Abstract

This paper introduces techniques to detect mutability of fields and classes in Java. A variable is considered to be mutable if a new value is stored into it, as well as if any of its reachable variables is mutable. We present a static flow-sensitive analysis algorithm which can be applied to any Java component. The analysis classifies fields and classes as either mutable or immutable. In order to facilitate open-world analysis, the algorithm identifies situations that expose variables to potential modification by code outside the component, as well as situations where variables are modified by the analyzed code. We also present an implementation of the analysis which focuses on detecting mutability of class variables, so as to avoid isolation problems. The implementation incorporates intra- and inter-procedural data-flow analyses and is shown to be highly scalable. Experimental results demonstrate the effectiveness of the algorithms.

1 Introduction

This paper introduces techniques to detect mutability of fields and classes in Java. We present a data-flow algorithm for this purpose. In our terminology, mutability is transitive, in that it covers modification not only of the memory location itself, but rather of all locations reachable from it. Special emphasis is placed on being able to analyze components, e.g. libraries, as opposed to analyzing whole programs only. In this aspect our techniques well fit recent challenges of software analysis as presented in

[19]. Java has spurred the development of component market, hence component analysis tools are expected to be of great practical importance.

We implemented the algorithms described in this paper. The Mutability Analyzer tool is shown to be highly scalable and can be applied to analyze libraries containing thousands of classes. In particular, we were interested in analyzing Java's runtime library which is fairly large, and represents several different coding styles. The results of applying our tool on this library are discussed later.

The discussion in the paper is in the context of Java, but it is also applicable to other object-oriented (OO) languages.

We have observed a wide spectrum of areas in which mutability information is applicable. Examples of such areas are security, code optimization, code analysis and software testing. The following elaborates on some important usages.

Isolation among applications running in the same JVM: The mechanism for sharing state in Java is via class variables. Isolation faults arise when composing components together, where one component depends on the state of a shared variable or object and another component changes that state. An actual implementation problem in the Java Development Kit (JDK) that occurred in version 1.1.1 was due to mutable static field sharing, and is described in [31]. As a result, an unprivileged applet was able to impersonate a trusted signature, causing a serious security fault. Identifying mutable class variables in the JDK can be used to avoid such isolation problems.

Distributed Shared Memory (DSM): An important factor in the efficiency of DSM is the proper distribution of objects between the processors [2]. Identifying immutable objects

* Java is a trademark of Sun Microsystems

[†] IBM Research Laboratory in Haifa, Israel

[‡] IBM T. J. Watson Research Center, NY, USA

allows the creation of duplicates on each of the processors in the DSM cluster. This eliminates the need to transport the objects from one processor to the other at considerable cost. In addition, information on immutable objects can be exploited to avoid the overhead of costly coherency and synchronization in multithreaded and distributed environments.

Concurrent Programming: Execution of concurrent programs on Symmetrical Multi-Processor (SMP) or Non-Uniform Memory Access (NUMA) systems introduces safety and liveness concerns. Designing for concurrency requires avoiding unsynchronized changes to shared state. Such synchronization introduces overhead costs. This overhead can be avoided for objects known to be immutable. An interesting example of a specific use of immutable objects identification is the generational garbage collector of Doligez et al. [11, 12] for ML [26]. This highly efficient garbage collector allocates only immutable objects in a local young generation subheap. The key idea is that immutable objects can be replicated without affecting program semantics. Our analysis can be used to identify immutable objects in Java, and therefore motivate the use of such generational garbage collector.

1.1 Mutability Analysis of Components

Our main contribution is a set of algorithms that incorporate flow-sensitive data-flow techniques for mutability analysis of Java components. The main complexity arises due to the reference semantics in Java, and the way in which we refer to changes in a variable state not only by locating direct assignments to it, but rather by detecting potential modifications to all the variables reachable from it. Moreover, our focus is on the analysis of software components (e.g. libraries or beans) rather than whole programs. This poses yet another level of complexity on our analysis, as the availability of all the accessors and modifiers depends on the analysis scope. Our solution is achieved by linking the notion of mutability to the notions of external accessibility and analysis scope.

The Mutability Analyzer tool is an efficient implementation of the algorithm presented in

this paper. The tool performs a static analysis on a given Java component, and classifies each field and each class in the component as *mutable* or *immutable*. As its output, the tool produces a list of mutability causes for those classes and fields in the component that are classified as *mutable*. The developers can use this information to modify their code so as to make certain fields immutable, or to avoid the potentially hazardous sharing of global state. To our best knowledge this feature is unique to our tool.

The paper is organized as follows. The next section provides the mutability definitions. Section 3 presents static analysis to determine mutability in the realm of component programming. The algorithm is described via a set of sub-analyses. Section 4 describes the Mutability Analyzer tool, shows experimental results, and presents our ideas of how to further enhance our tool. Section 5 outlines related work. Section 6 concludes with open issues.

2 Mutability in Java

In this section we introduce definitions of states in Java programs. Since our analysis acts on classfiles, we will use the classfile terminology [24] to address mutability issues.

2.1 Variable State and Object State

An object in Java is either a class instance or an array. Object instantiation involves creation of a set of instance variables or array components, respectively. Thus, an object represents an aggregation of associated variables.

A class variable in Java corresponds to a field declared within a class declaration using the `static` modifier. Note that fields declared within an interface declaration are necessarily defined `static`. In contrast, an instance variable corresponds to a field declared within a class declaration without the `static` modifier. A class instance is an aggregation of variables corresponding to non-`static` fields declared in this class and in all of its ancestors. We say that a class type *implements* a non-`static` field if its instances contain a variable associated with

that field.

For primitive types, the state of a variable is defined by the primitive value which it holds. However, a variable holding a reference to an object is more complex, as it may indirectly refer to other variables. We formally distinguish between the following two notions.

- *Value-state of a variable* is the value held in the variable.
- *State of a variable* is defined by the set of all the value-states of the variables that are reachable from it.

Obviously, the state of a primitive variable coincides with its value-state.

State of an object is defined by the set of states of all its associated variables. Thus the state of a reference type variable, whose value-state is non-null, is recursively defined by the variable's value-state together with the state of the referenced object.

2.2 Immutability Definitions

Every variable is implicitly initialized with a type-specific default value when its storage is allocated. Defining immutability as total absence of modifying code would classify as immutable only variables that are never set by the user code. Hence we define immutability as absence of modification code that can be executed after a certain execution point, when the variable or the object is considered to be "fully created". We refer to this point as the *initialization point* of the variable or the object. The initialization point of a class variable is upon completion of its corresponding static initializer, i.e. the class `<clinit>` method. Similarly, the initialization point of an instance variable or a class instance is upon the completion of a corresponding constructor, i.e. an `<init>` method. The initialization point of an array object or its components is upon execution of a corresponding array creation bytecode instruction, i.e., `newarray`, `anewarray`, or `multianewarray`.

We say that *a variable or an object is immutable* if and only if its state never changes after the corresponding initialization point.

In order to define immutability of a field, we make use of the fact that any class or instance variable corresponds to some field. We say that *a field is immutable* if and only if all the variables that correspond to that field are immutable.

Next, we proceed with immutability of a class type. Since the state of a class instance is determined via its non-`static` fields, we say that *a class is immutable* if and only if all non-`static` fields implemented by it are immutable¹.

For our purposes, the state of an object is determined through the variables that correspond to non-`static` fields only. Thus, an object like an instance of `java.lang.Object`, which has no non-`static` fields, is defined to be immutable. Note that the JVM may implicitly attach other storage locations to the object that are not considered to be part of its state according to our definitions. One such implicit variable is the lock associated with every Java object.

2.3 Access Control and Immutability

Java and the bytecode provide basic means for controlling access to variables and their referenced objects via access modifiers: `private`, `public` and `protected`. These modifiers restrict the visibility of classes and their members and consequently the ability to modify their states.

The `final` access modifier provides partial support to ensure value-immutability of variables. The language forbids assignments to variables that correspond to a `final` field, except within the initializer. Thus their value-states cannot be modified after the initialization point. However, if such a variable holds a reference to a mutable object, then the state of the referenced object may be changed, although the variable will always refer to the same object.

¹Since `abstract` classes and interfaces cannot be instantiated, their mutability is not considered here.

3 Mutability Analysis

3.1 Analysis of Software Components

According to our definitions, a variable would be identified as immutable if it can be shown that there are no methods which may modify its state. Availability of all the modifiers/accessors of a particular variable depends on the analysis scope. We distinguish between the two following extremes:

- A *closed-world analysis scope* where all symbolic references refer to analyzed classes, and all possible targets of runtime invocations are defined in analyzed classes.
- An *open-world analysis scope* (or *component analysis*) where the analyzed component is an arbitrary collection of classes.

In closed-world processing, mutability analysis would identify the set of direct modifiers of variable states. In contrast, an open-world mutability analysis should also identify the situations that expose variables to potential state modification by code outside the analysis scope, and conservatively classify them as mutable. Our analysis accepts any kind of analysis scope and considers a closed-world processing as a degenerate case of an open-world analysis. Thus, from here on we concentrate on an open-world analysis.

We define the notion of variable and object accessibility so as to facilitate the open-world analysis. We say that a *variable is value-accessible* if its value may be modified by code outside the analysis scope. A *variable is state-accessible* if a variable reachable from it is value-accessible. Note that external references to immutable objects do not affect state-accessibility since by our definition accessibility is only concerned with mutable state.

Mutability analysis should thus handle a spectrum of situations that may cause variable mutability, some of which are not trivially perceived. The example in Figure 1 illustrates several such situations.

3.2 High-level Mutability Analysis Algorithm

Next, we introduce our data-flow algorithm that performs conservative static component analysis. We basically distinguish between two major parts within a mutability analysis:

- State modification analysis that is used to determine possible modification of a variable's state by methods in classes within the analysis scope.
- State accessibility analysis that is used to determine possible modification of a variable's state from outside the analysis scope, i.e. by methods defined in classes that are not part of the analyzed component.

The scope of the analysis is identified by a given set of classfiles forming a Java component. Each classfile corresponds to an analyzed class or interface. The scope is open, so a given classfile may include references to classes or interfaces whose classfiles are outside the analysis scope.

Conservative assumptions are applied whenever the analysis requires unavailable cross-class global information. Classes outside the analysis scope are assumed to be mutable. Classes that extend classes which are outside the analysis scope are assumed to be mutable because they may inherit mutable instance variables. Similarly, whenever an analyzed method contains a method invocation (`invokestatic`, `invokevirtual`, `invokespecial`, `invokeinterface`), the analysis applies conservative assumptions when it decides that there might be a target implementation residing outside the analysis scope.

As stated above, a non-`abstract` class is immutable if all the non-`static` fields it implements are immutable. Moreover, determination of a (reference-type) variable's state-accessibility is dependent on the known set of immutable classes. For example, if the class variable that corresponds to a `public final static` field may refer to an instance of a mutable class, then it would be considered as state-accessible and therefore mutable. Thus, identifying the mutability of non-`static` fields and of

```

public class Sample {
    /*****
        Fields accessible from outside the component
        *****/
    public Object anObject;           // variable and referenced object are accessible
    public final int[] anArray = {1,2,3}; // referenced object is accessible
    /*****
        Private fields
        *****/
    private Vector privateData;
    /*****
        Constructors causing accessibility of the object referenced by privateData
        *****/
    public Sample() {
        privateData = new Vector();
        privateData.add(anArray);      // anArray is an accessible variable
    }
    public Sample(Object data) {
        privateData = new Vector();
        privateData.add(data);        // data may be referenced from outside
    }
    /*****
        Methods causing modification of privateData
        *****/
    public void resetData() { privateData = new Vector(); } // value is modified
    public void removeData() {
        privateData.removeAllElements(); // value unchanged, state modified
    }
    /*****
        Methods causing accessibility of privateData
        *****/
    public void addData(Object data) {
        privateData.add(data);        // parameter becomes part of the state
    }
    public Object[] getData() {
        return privateData.toArray(); // a mutable part of the state is returned
    }
    public boolean isEqual(Vector v) {
        return v.equals(privateData); // passing part of state to a method outside the scope
    }
    public void exposeData() {
        anObject = privateData;      // aliasing part of state with an accessible variable
    }
    public void exposeData(Object[] array) {
        array[0] = privateData.elementAt(0); // aliasing with a parameter
    }
    public Object[] exposeData(int i) {
        Object[] array = new Object[1];
        array[0] = privateData.elementAt(i);
        return array;                // aliasing with a returned object
    }
}

```

Figure 1: An example for a spectrum of situations that cause variable mutability

classes is interdependent. Therefore, the analysis requires iterative processing. Each iteration determines immutability of non-`static` fields (and thereby immutability of classes) based on a set of already determined immutable classes, until we reach a fixed point. If there are classes whose immutability is not determined during the iterative process, they are conservatively considered to be mutable. In contrast, identifying the mutability of `static` fields (class variables) does not affect mutability of classes, and is computed after classifying all the classes as either mutable or immutable.

Each analyzed non-`abstract` class and each of its implemented fields are classified during the course of our algorithm as *mutable*, *immutable*, or *undecided*. The *undecided* classification indicates that further analysis may eventually change the classification to *mutable* or *immutable*.

3.2.1 Determining Mutability of a Field

The routine *TestField* is used in the algorithm to determine mutability of a given *undecided* field, based on a set of *mutable*, *immutable* and *undecided* classes. The input field is specified by its name and the declaring class². The routine uses the information on class mutability (as derived from the classes' classification) to set the classification of the given field to be *mutable* or *immutable*, or leave it as *undecided*. It may occur that a field cannot be classified as *immutable*, but could be classified as such if more of the classes currently classified as *undecided* were reclassified as *immutable*. In the case of insufficient class mutability information, a field's classification remains *undecided*. If there is no non-`abstract` class classified as *undecided*, then upon completion of *TestField* the field is classified either as *mutable* or as *immutable*³.

In order for the routine to determine the mutability of the given *undecided* field, it refers to the initialization point that correspond to that field, i.e. to the `<clinit>` or to the appropriate `<init>`.

²For a non-`static` field, the implementing class is also provided.

³*TestField* is invoked for `static` fields when all classes are already classified as *mutable* or *immutable*.

The structure of the *TestField* routine is outlined in Figure 2.

3.2.2 Determining Mutability of every Field and Class in a Component

Next we describe the main iterative processing that uses the *TestField* routine to establish mutability of classes and fields. The algorithm starts with a given set of *mutable*, *immutable*, and *undecided* classes and fields. For example, the (widely used) class `java.lang.String` requires native code analysis in order to properly establish its immutability. Thus, if this class is part of the analysis scope, we generally expect it to be initially classified as *immutable* so as to get more accurate results for other classes. Upon completion of the algorithm, every class and every `static` field is classified either as *mutable* or as *immutable*. The algorithm appears in Figure 3.

4 The Mutability Analyzer Tool

In this section we describe the Mutability Analyzer tool that we developed to perform static mutability analysis. The tool performs an open-world analysis on a given Java component, and classifies each `static` field and each class in the component as *mutable* or *immutable*. As its output, the tool produces a list of mutability causes for those classes and fields in the component that are classified as *mutable*. In particular, for each `static` field, the tool reports a list of conditions (A-D), as defined in *TestField*, that do not hold for this field, along with additional information.

4.1 *TestField* Implementations

A primary objective of our tool is to run it on very large components. The implementation is designed with a special emphasis on scalability. As a result, different *TestField* routines were developed, one for `static` fields and the other for non-`static` fields. The two implementations of *TestField* differ in the analyses they employ to test conditions A-D, as specified in Figure 3. Table 1 describes these differ-

Routine *TestField*

Input: Set of classes, each classified as *mutable*, *immutable*, or *undecided*; field *Afield* classified as *undecided*

Output: Classification of *Afield* as *mutable*, *immutable*, or *undecided*

Remark: if there are no *undecided* classes, *Afield* should be classified as *mutable* or *immutable*

1. test A: Value modification //value-state may be modified after initialization point
2. **if true, return** *mutable*
3. test B: Object modification //state of the referenced object may be modified after initialization point
4. **if true, return** *mutable*
5. test C: Value accessibility //value-state may be modified from outside the analysis scope
6. **if true, return** *mutable*
7. test D: Object accessibility //state of the referenced object may be modified from outside the analysis scope
8. **if D depends on an undecided class return** *undecided*
9. **if true, return** *mutable*
9. **return** *immutable*

Figure 2: Determining immutability of a field

Input: Initial set of classes and fields, each classified as *mutable*, *immutable*, or *undecided*

Output: Classification of each class and each **static** field as *mutable* or *immutable*

1. **do**
 - /* determine mutability of classes and non-static fields */
 - 2. **for** every non-abstract class classified as *undecided*
 - /* determine mutability of non-static fields implemented by the class */
 - 3. **if** the full list of non-**static** fields implemented by the class is unknown
 - /* there might be a mutable non-**static** field implemented by the class */
 - 4. classify the class as *mutable*; skip to the next class
 - 5. **if** there exists a non-**static** mutable field implemented by the class
 - 6. classify the class as *mutable*; skip to the next class
 - 7. **for** each non-**static** *undecided* field *Afield* implemented by the class
 - 8. *TestField(Afield)*
 - 9. **if** *Afield* is classified as *mutable*
 - 10. classify the class as *mutable*; skip to the next class
 - /* all non-**static** fields implemented by the class are either *undecided* or *immutable* */
 - 11. **if** all non-**static** fields implemented by the class are *immutable*
 - 12. classify the class as *immutable*
 - 13. **until** the set of classes classified as *undecided* has not been reduced in the current iteration
 - /* reached a fixed point */
 - 14. **for** every class classified as *undecided*
 - 15. classify the class as *mutable*
 - /* determine mutability of static fields */
 - 16. **for** every analyzed class or interface
 - 17. **for** every **static** *undecided* field *Afield* declared within the class
 - 18. *TestField(Afield)*

Figure 3: Determining immutability of a component

ences. The *TestField* routine for `static` fields employs more complex sub-analyses, some of which require inter-procedural iterations.

For efficiency reasons, we perform a sequence of sub-analyses, each processing the whole code and extracting information per each analyzed method. Each sub-analysis accumulates information for all the relevant fields. A sub-analysis is activated when its functionality is required for the first time; this information is reused during consecutive invocations of *TestField*.

The Mutability Analyzer implementation uses core libraries that were implemented as part of *Toad* [33]. Toad is a post production environment that allows for a symbiosis of dynamic information about a running application, with `static` information gathered from the classes that comprise it. The Toad environment has been developed as an umbrella framework for a suite of core libraries and tools that monitor, understand and optimize Java applications. Particularly, the CFParse library [7] allows the user to read and write classfiles as well as edit them. The JAN library [29] collects and manipulates static information about a Java component (e.g. application, applet or servlet) by analyzing a set of classfiles, and effectively constructing the component's reference, hierarchy and call graphs.

On top of the abstractions that are provided by CFParse and JAN, we implemented additional two core libraries, each being an engine for data-flow analysis.

- *Intra-procedural engine* that is used to iteratively compute the effect of an instruction on information associated with locations on the method frame (operand stack and local variables array).
- *Inter-procedural engine* that is used to compute the effect of a method on information associated with the variables escaping this method.

Whenever a callee method may reside outside the analysis scope, the client of the inter-procedural engine should make a conservative estimate on the effect of this method. Currently the tool assumes that any virtual call (`invokevirtual` or `invokeinterface`) may

have a potential target implementation outside the analysis scope. This is a major source of conservativeness for the analysis. Section 4.3 discusses potential enhancements to the tool.

In addition to the above core libraries, we implemented a set of utility analyses.

- *Type analysis* that identifies, for each method instruction, the set of possible run-time types of its operands.
- *Reachability analysis* that identifies, for each method instruction, the reachability relationships of its operands.

These utilities provide the basis for the sub-analyses of the Mutability Analyzer, which perform the tests described in the algorithm on Figure 2. For example, state modification analysis would involve identification, for each object modifier instruction (`putfield`), the reachability relationships of the modified object.

Our analysis does not process native code, nor does it take into account dynamic effects resulting from reflection.

4.2 Results

We evaluated the results of our tool by comparing them to the results derived by a *reflection-based tool*. The latter tool does not require any processing of method bodies. It considers any non-`final` field to be value-mutable, and any field whose declared type is mutable to be state-mutable.

We use the runtime library *rt.jar* from the Java 2 JDK release 1.2 to illustrate the benefits of our approach. We chose this library since *rt.jar* is fairly large (4239 classes, 35999 methods), and represents a reasonably diverse set of coding styles.

Figure 4 gives the distribution of mutable and immutable `static` fields for the two algorithms. Both algorithms were run with the class `java.lang.String` being classified as *immutable* as input to the analysis.

The graph shows growth in the number of fields identified as *immutable* when the Mutability Analyzer is used. Note that identification of primitive `final static` fields as *immutable* is simple and they are treated in the same way by any tool which ignores violation

Condition	TestField
A: Value modification Non-statics	(1) A method, which is not a declaring class instance initializer <code><init></code> , has a <code>putfield</code> that refers to that field, or (2) A <code>putfield</code> in an <code><init></code> that refers to that field affects an object different from the <code>this</code> object
Statics	A method, which is not the declaring class initializer <code><clinit></code> , has a <code>putstatic</code> that refers to that field
B: Object Modification Non-statics	A possible runtime type of a variable corresponding to the field is mutable
Statics	(1) A possible runtime type of a variable corresponding to the field is mutable, or (2) A method may modify the state of the object referenced by the class variable
C: Variable Accessibility Non-statics	Field is non-private and non-final
Statics	Field is non-private and non-final
D: Object Accessibility Non-statics	A possible runtime type of a variable corresponding to the field is mutable
Statics	(1) A possible runtime type of a variable corresponding to the field is mutable, or (2) The field is non-private, or upon a <code>putstatic</code> that refers to that field, a mutable object reachable from the class variable may be referenced from a non-local variable, or (3) A method may create non-local references to a mutable object reachable from the class variable

Table 1: *TestField* variations in the Mutability Analyzer tool

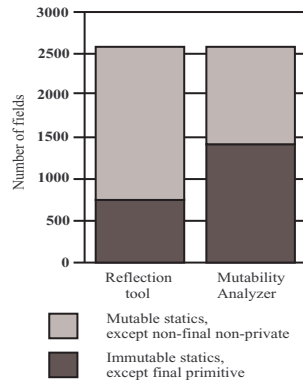


Figure 4: Distribution of *mutable/immutable static* fields in *rt.jar*

of Java access rules from native code or via reflection. Non-final non-private static fields would be identified as *mutable* in every analysis which does not take into account runtime accessibility constraints. Therefore, the improvement comes from reducing the number of other mutable static fields. We intend to enhance

our tool so that the sizes of both categories of mutable static fields will be further reduced.

In addition to the classification of fields as *mutable* or *immutable*, the Mutability Analyzer provides the information on mutability causes, such as location of the potentially modifying code. The developers can use this information to modify their code so as to make certain fields immutable, or to avoid the potentially hazardous sharing of global state. To our best knowledge this feature is unique to our tool.

The Mutability Analyzer tool was also applied on a large internal IBM framework (4634 classes, 36363 methods), containing multiple libraries, applets, and servlets. This framework contains 3553 static fields, of which 2324 are compile-time constants. Of the remaining 1229 fields, 992 fields are *mutable* either because they are non-final and non-private or because their value-state is modified. Of the remaining 237 fields, more than two thirds (160 fields) are identified as *immutable*.

One of our primary concerns during the implementation of the tool was its scalability. Of particular concern were the sub-analyses which

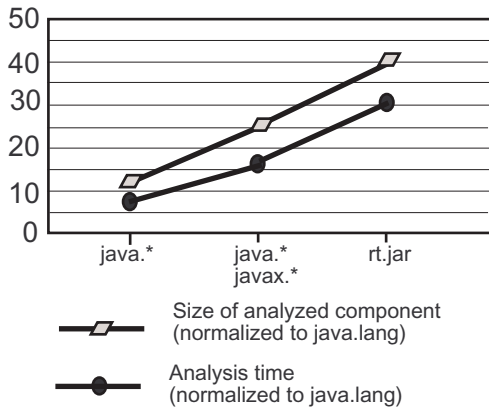


Figure 5: Efficiency of the Mutability Analyzer

require inter-procedural iterations: reachability analysis, state accessibility analysis, and state modification analysis. In practice, the tool proved to be fairly scalable. The tool was run on a Pentium III 500 with 128 Mb RAM, using the Sun JVM 1.3.0rc1-T. The analysis time for the full *rt.jar* is approximately 20 minutes. Although the analysis requires interprocedural processing, in practice it is inherently efficient and almost linear in the problem size. This can be seen from Figure 5 that shows the results of the analysis on different parts of *rt.jar*.

4.3 Tool Enhancements

As previously stated, the analysis is designed to primarily support open-world analysis. In the general case, this entails that all non-private fields and methods are considered as accessible from outside the analysis scope. In practice, however, there are certain circumstances in which accessibility of fields and methods may become more restrictive. For example, the component may be perceived as a Java application that is expected to run as the sole component in a JVM. In that case, no methods should be considered as being accessible from outside the analyzed component, other than those that can be accessed from the JVM (e.g., the `main` method from which the application starts running, or methods overriding JDK methods). Such information would sig-

nificantly reduce the size of the component's call-graph. Consequently, the amount of mutability causes reported by the Mutability Analyzer would be substantially reduced, and more fields would be identified as immutable. In general, additional information on those methods from which the component can start its execution would reduce the set of accessible methods. Such information can be supplied to the Mutability Analyzer as meta data. An effective construction of a component call-graph that uses such meta data (as *export specification*) is implemented in Toad [33].

The run-time resolution process that determines the actual instance variable or class variable that gets accessed is subject to runtime access restrictions. It is the `SecurityManager` that is responsible for restricting the run-time accessibility. A particular runtime access restriction is derived from package `sealing`, i.e. a certain characterization of a package telling that all the classes in it must come from the same JAR file. Similarly to the case of the meta-data discussed above, information on sealed packages would restrict the accessibility abstraction of fields and methods. We expect that most of the 541 non-private non-final *rt.jar* fields that are currently determined as mutable, would be determined as inaccessible from outside the component, since all but two packages of *rt.jar* are considered as being sealed. We plan to implement a generic module which would use these two kinds of accessibility information in future versions of the Mutability Analyzer.

We observed that a major source for over-conservativeness in the current version is derived from the fact that the analysis assumes that each virtual invocation may be resolved to a method that is defined outside the analyzed component. This implies that arguments passed at such invocations are considered to be potentially modified. We plan to optimize the component call-graph in the sense of identifying certain virtual call sites for which all possible target implementations reside within the analysis scope. We currently investigate usage of our type analysis utility for this kind of optimization. In addition, information on accessibility restrictions, as discussed above, may be further used for better call resolution. Sealing

information was used in [35] to identify sealed calls, i.e. calls whose potential targets are confined to a package and can be completely determined at analysis time. It has been shown that in about half of the sealed packages in `rt.jar`, 5-60% of the virtual calls to `non-final` methods are identified as sealed.

5 Related Work

Data-flow algorithms generally use `static` abstractions of dynamic behavior. Alias analysis or point-to analysis [13, 15, 16] use abstractions of dynamic data structures that include pointers (like in C or C++) or references (like in Java). These abstractions capture connectivity relationships between variables. For example, Ghiya and Hendren [15] present a pointer analysis to detect invariant variables over a region, and use this information to optimize code. In OO programming the analyses determine properties on reachability relations among objects as well as among variables. In contrast to the work presented here, most of the data-flow algorithms that appear in the literature require that the whole program be available for the analysis (like in [32]).

Several works that have been published recently [4, 5, 34, 8] propose data-flow algorithms to determine if objects can be allocated on the stack, and if they are accessed only by a single thread. Choi et al. introduce in [8] a useful static abstraction for Java programs, connection graph, that captures the connectivity relationships between objects. Our reachability analysis is simpler since we summarize the reachability relations for sets of variables, rather than per each variable. We chose to implement such thinner version for scalability reasons.

Jackson and Rinard [19] describe the challenges of software analysis by presenting a series of dichotomies. They argue that, in the future, analyses will be modular and incremental to enable analysis of components. Their view is that analyses will be focused and partial, rather than uniform, paying closer attention to specific parts of the software. We believe that the analyses that we develop in the Mutability Analyzer, fit well in these dichotomies.

In traditional procedural languages, such as Pascal and C, the state of a running program is the composition of all values held in the program variables. In these languages the variables are either primitive (e.g., integer and float) or compound (e.g., arrays and structures). A change to a variable's state is detected by identifying direct or indirect assignments to that variable. However, problems arise whenever there are indirect assignments to the variable. OO programming, such as C++ and Java, introduces additional levels of program state via abstractions such as fields, objects and classes. A change to an object's state is not specified only by assignments to its variables, but also by modifications on the states of objects to which these variables refer. The complete state of an object becomes more complicated [17]. This reference semantics has been the focus of many works that address ways to control aliasing and sharing of state and to improve locality [1, 14, 17].

Our work was originally motivated by observing the need to identify immutability of `static` fields in order to ensure isolation properties while composing Java components. Lack of isolation poses serious integrity, scalability and security problems. In C++, the `const` specifier can be applied to variables, objects, arrays and methods, and implies logical and physical const-ness. This enforcement is discussed in the context of program design in [23]. In some languages, immutability of types and/or variables is part of the language. For example, Sather [28] and ML [26] allow types to be defined as immutable so as to facilitate stack allocation of objects and more effective garbage collection in the realm of multithreaded environments [11, 12]. ML defines also immutable variables, so that the isolation concerns are addressed as well. Similar isolation concerns motivate works on flexible alias protection [27, 9, 6]. Bokowski and Vitek [6] present a set of syntactic constraints that strengthen encapsulation in OO programs, and consequently facilitate the implementation of secure systems. Such confinement is technically achieved by extending Java with two additional modifiers, one for classes and one for methods. Our work attempts to guarantee, by applying a data-flow analysis on any Java component

and without any language or JVM extensions, that confinement is respected. ACE (Access Controlled Encapsulation) [21] and JAC (Java with Access Control) [22], address accessibility and isolation issues in OO languages. ACE presents a uniform model of access rights that overcome the limitations of `const T*` in C++. JAC focuses on practical usability by extending the Java language so as to support `const`-ness. ESC (Extended Static Checking) [10] accepts an annotated program and checks for certain programming errors. Part of a method’s annotation is a list of modified variables. However, such a list is not always available at compile time. Our approach is a conservative variant of that introduced in ESC. Another related work [30] introduces data groups to facilitate annotation-based verification of value modification in presence of subclassing and method overriding.

An interesting work based on specifying global security properties pertaining to the control-flow of a whole program appears in [20]. They provide a formal framework for the definition of such properties, and propose a technique for their verification based on finite-state model-checking.

A different research area that addresses the complexity of manipulating references and mutable data is investigated via formal logical systems. Mason and Talcott [25] present a formal system for reasoning about equivalence of Lisp or Scheme programs that act on objects with memory. Most subtleties of their language arise due to the presence of mutable data. This work is further described and enlarged in [18]. The formal extended logic is called VTL_{oE} (Variable Typed Logic of Effects) and it goes beyond traditional programming logics. It has a relatively complete axiomatization of primitives for mutable data as well as a variety of induction principles and methods for proving properties of programs.

Security faults due to Java’s semantics on inner classes is the focus of the work done by Pugh and Bhowmik [3]. Inner classes were added to Java in version 1.1, in a way that was compatible with version 1.0 of the JVM. Given two classes A and B, if one is declared inside the other, or if both are declared inside another class, then A and B should have access to each

other’s `private` fields and methods. Since the JVM does not directly support inner classes, the compiler produces a modified program in which the `private` protection of the fields and methods has been essentially eliminated. Pugh and Bhowmik developed a technique for closing this security hole by changing the classfile (via additional parameters and instructions in the accessor methods) that enforce the contract between an inner class and its containing class.

6 Conclusions and Open Issues

We presented definitions for mutability in Java that can serve as the basis for further research work in this area, and illustrate an innovative approach for static analysis in order to automatically detect mutable and immutable variables, fields, objects, and classes. One of the major contributions is in coping with an open-world analysis, thus being able to accept any Java component as the analysis scope. Our Mutability Analyzer tool demonstrates the strength of our approach, and examines our definitions. Despite the fact that an open-world analysis has been employed, the prototype successfully categorizes class variables as well as instance variables and classes for the Java runtime.

One of the major design decisions that drove our implementation was to use basic core libraries such as CFParse and JAN, and introduce general-purpose engines for intra-procedural and inter-procedural analyses. The code is designed to be scalable and fit into a multi-level static analysis framework, so that utilities and sub-analyses can be used and extended to deal with properties other than mutability characterization.

We realize that static analysis is in some cases limited. Therefore, for properties that the analysis will not be able to detect statically, we plan to facilitate smart annotations so as to detect those cases at run time. This can be done by using the CFParse core library to parse, edit and annotate classfiles.

We also explore future directions for the analysis expansion, such as interval immutability analysis to determine immutability of vari-

ables at certain intervals, e.g., during the invocation of a specific method, and modular immutability analysis which would allow plugging together the results of mutability analysis of sub-components to obtain analysis of the full component. We wish to enhance the definition of immutability starting from a certain initialization point so as to cover lazy initialization, i.e., cases where a variable is set only once but not necessarily during the class or instance initialization.

We believe that the Mutability Analyzer and its reports can serve as a useful source of information for Java developers, as well as for JVM architects, to better address security issues, code optimization, code analysis and software testing.

Acknowledgements

We would like to thank Guy Laden, Ayal Zaks and Yossi Gil for helpful comments and useful technical discussions. Thanks go also to Erez Petrank for introducing us to the area of garbage collection, where immutability information can be usefully applied. Special thanks go to Dafna and Jacob Sheinwald.

About the Authors

Sara Porat received her Ph.D. in Computer Science from the Technion, Haifa, in 1986. She has been a faculty member of the University of Rochester and the Technion. Since joining IBM Israel in 1990, she has been involved with language extensions, compilers and environment tools. She can be reached at: IBM Haifa Research Lab, MATAM, Haifa, Israel 31905. Her Internet address is porat@il.ibm.com.

Marina Biberstein received her M.Sc. degree from the Technion - Israel Institute of Technology in Haifa, in 1999. She joined IBM Haifa Research Laboratory in 1999. Her research interests include program analysis and error-correcting codes. She can be reached at: IBM Haifa Research Lab, MATAM, Haifa, Israel 31905. Her Internet address is biberstein@il.ibm.com.

Larry Koved received his M.Sc. in Computer Science from the University of Maryland, College Park in 1985. He joined IBM T.J. Wat-

son Research Center in 1982. He is currently the co-lead for the IBM Java Security team. His research has included hypertext, replication algorithms, synchronous multi-user collaboration, Virtual Reality environments, mobile computing, user interface technology, object technology and Java security. He can be reached at: IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, N.Y. 10598. His Internet address is koved@watson.ibm.com.

Bilha Mendelson received her Ph.D. in Electrical and Computer Engineering from the University of Massachusetts at Amherst, in 1990. She joined IBM Haifa Research Laboratory in 1990 and manages the Code Optimization group. Her main areas of interest are: code optimization, compiler optimization and modern architecture. She can be reached at: IBM Haifa Research Lab, MATAM, Haifa, Israel 31905. Her Internet address is bilha@il.ibm.com

References

- [1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Aksit and S. Matsuoka, editors, *Lecture Notes in Computer Science*, volume 1241, pages 32–59, Jyvaskyla, Finland, June 1997. ECOOP'97, 11th European Conference on Object-Oriented Programming.
- [2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single-system image of a JVM on a cluster. In *Proceedings of the IEEE 28th International Conference on Parallel Processing*, pages 4–12, Aizu-Wakamatsu, Fukushima, Japan, September 1999.
- [3] A. Bhowmik and W. Pugh. A secure implementation of Java inner classes. <http://www.cs.umd.edu/~pugh/java/#sic>.
- [4] B. Blanchet. Escape analysis for Object Oriented languages. Application to Java. In *Proceedings of the 1999 Conference On Object-Oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, November 1999.

- [5] J. Bogda and U. Holzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 Conference On Object-Oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, November 1999.
- [6] B. Bokowski and J. Vitek. Confined types. In *Proceedings of the 1999 Conference On Object-Oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, November 1999.
- [7] CFParse. <http://www.alphaworks.ibm.com/tech/cfparse>.
- [8] J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 1999 Conference On Object-Oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, November 1999.
- [9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices - Proceedings of the 1998 Conference On Object-Oriented Programming, Systems, Languages, and Applications*, volume 33(10), pages 48–64, October 1998.
- [10] D. Detlefs, K. Rustan, M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, CA, December 1998.
- [11] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *ACM SIGPLAN Notices - Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, 1994.
- [12] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *ACM SIGPLAN Notices - Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, January 1993.
- [13] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural point-to-analysis in the presence of function pointers. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 121–133, San Diego, California, January 1998.
- [14] D. Genius, M. Trapp, and W. Zimmermann. An approach to improve locality using sandwich types. In *Proceedings of the Second Types in Compilation Workshop*, volume LNCS 1473, Kyoto, Japan, March 1998.
- [15] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Proceedings of the 1994 SIGPLAN Conference On Programming Language Design and Implementation*, Orlando, Florida, June 1994.
- [16] M. Hind, M. Burke, P. Carini, and J.D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.
- [17] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. W3: The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
- [18] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, May 1995.
- [19] D. Jackson and M. Rinard. *The Future of Software Engineering*, chapter The Future of Software Analysis. ACM Press, June 2000.
- [20] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. In *Proceedings of the 20th IEEE Security and Privacy Symposium*, Oakland, California, 1999.
- [21] G. Kniesel. Encapsulation = visibility + accessibility. Technical Report TR-96-12, CS Dept., University of Bonn, Germany, 1996.
- [22] G. Kniesel and D. Theisen. JAC - Java with transitive readonly access control. In

Proceedings of the Intercontinental Workshop on Aliasing in Object-Oriented Systems, Lisbon, Portugal, June 1999.

Conference On Object-Oriented Programming, Systems, Languages, and Applications, Denver, Colorado, November 1999.

- [23] J. Lakoš. *Large-Scale C++ Software design*. Addison-Wesley, 1996.
- [24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
- [25] I. A. Mason and C. L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.
- [26] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
- [27] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *Proceedings of 1998 European Conference on Object-Oriented Languages*, Brussels, Belgium, July 1998.
- [28] S.M. Omohundro. The Sather programming language. *Dr. Dobbs' Journal*, 18(11), October 1993.
- [29] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with Java. In *Proceedings of CASCON'98*, Toronto, Canada, November 1998.
- [30] K. Rustan and M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, October 1998.
- [31] Secure Internet Programming Group at Princeton University. <http://www.cs.princeton.edu/sip/news/april29.html>.
- [32] V. Seshadri. IBM High Performance Compiler for Java. *AIXpert Magazine*, September 1997.
- [33] Toad. <http://www.alphaworks.ibm.com/tech/toad>.
- [34] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 1999*
- [35] A. Zaks, V. Feldman, and N. Aizikowitz. Sealed calls in Java packages. Accepted for publication at the 2000 Conference On Object-Oriented Programming, Systems, Languages, and Applications.