Precise Identification of Side-effect-free Methods in Java

Atanas Rountev
Department of Computer Science and Engineering
Ohio State University
rountev@cis.ohio-state.edu

Abstract

Knowing which methods do not have side effects is necessary in a variety of software tools for program understanding, restructuring, optimization, and verification. We present a general approach for identifying side-effect-free methods in Java software. Our technique is parameterized by class analysis and is designed to work on incomplete programs. We present empirical results from two instantiations of the approach, based on Rapid Type Analysis and on points-to analysis. In our experiments with several components, on average 22% of the investigated methods were identified as free of side effects. We also present a precision evaluation which shows that the approach achieves almost perfect precision—i.e., it almost never misses methods that in reality have no side effects. These results indicate that very precise identification of side-effect-free methods is possible with simple and inexpensive analysis techniques, and therefore can be easily incorporated in software tools.

1 Introduction

Side effects of a method are state changes that can be observed by code that invokes the method. The presence or absence of side effects is an important property that has a variety of uses in software tools. Since side effects have negative effect on program comprehension [7], valuable programmer time can be saved by using a program understanding tool that automatically identifies and labels side-effect-free methods. Such a tool can simplify the use of a method, by eliminating the need to investigate manually the actions performed in the method body and in all of the method's transitive callees.

The absence of side effects is a desirable property that is often defined at design time. For example, UML defines an attribute `isQuery` for operations. This attribute specifies whether the operation "leaves the state of the system unchanged (`isQuery` = true) or side effects may occur (`isQuery` = false); the default value is false" [19]. In reality, design and code often diverge. For example, in modern iterative and incremental development it is often necessary to reverse-engineer the design from existing code. A typical scenario is to perform design recovery through reverse engineering of the last iteration's code in the beginning of the current iteration [12]; the results serve as the starting point for subsequent design work. In this case, identification of side-effect-free methods in the code can ensure consistency with the `isQuery` properties from the design.

In optimizing compilers and program transformation tools, the absence of side effects allows a number of semantics-preserving transformations. Similarly, the correctness of some refactorings [8] depends on ensuring that methods do not have side effects. Investigating the differences between the sets of side-effect-free methods before and after code modifications can also provide a simple form of change impact analysis.

Information about side-effect-free methods is also useful in other situations. For example, in query-based debugging for object-oriented programs (e.g., [14]), it is necessary to know that a query expression does not have side effects. Similarly, expressions in assertions (e.g., in JML [13]) should be free of side effects. In some approaches for software security (e.g., SecureUML [17]), permissions can be restricted to allow only invocations of side-effect-free methods. In all these cases, it is necessary to be able to determine precisely which methods do not have side effects.

Side-effect analysis determines which memory locations may be modified by the execution of a program statement. Unfortunately, existing work on side-effect analysis cannot be used directly to identify side-effect-free methods. First, most of this work is for procedural languages, while our goal is to analyze Java code, which requires handling of virtual calls. Second, side-effect analyses are typically designed to analyze complete programs. However, in the context of software tools, it is essential to be able to perform separate analysis of software components. For example, it is typical to have to analyze a component without having access to the clients of that component. Whole-program side-effect analysis cannot solve this problem.

Another problem with existing work on side-effect analysis is that it does not provide information about the degree of imprecision in the analysis results—that is, how often does the analysis make "false claims"? This is a critical issue for the use of program analysis in software tools, because imprecision may lead to waste of human time and effort, and possibly render the analysis useless.

The goal of this work is to develop analysis techniques for identifying side-effect-free methods in incomplete Java programs, and to evaluate the imprecision of these techniques. We describe a static analysis which identifies methods that are free of side effects with respect to the clients of the analyzed component. The approach is parameterized by *class analysis*, which determines the classes of all objects to which a reference variable or a reference field may point. We use class analysis to identify the calling relationships between methods and to characterize the heap objects in the modified state. Our approach shows how a large body of work on class analysis can be used to discover side-effect-free methods in incomplete programs.

We have implemented two instances of this approach, one based on Rapid Type Analysis (RTA) [2] and the other based on context-sensitive points-to analysis [18]. Since these two techniques belong to opposite ends of the cost/precision spectrum, this investigation provides insights that are applicable to a large portion of the design space for the problem. For our subject components, the approach based on [18] achieves *perfect precision*—that is, it discovers all side-effect-free methods. Furthermore, the RTA-based approach achieves almost perfect precision. In both cases, a large number of methods (22% on average) are identified as having no side effects. These results indicate that (1) side-effect-free methods occur often, and (2) such methods can be identified very precisely with inexpensive analysis techniques.

This work has the following contributions:

- We propose a parameterized analysis for identifying side-effect-free methods in incomplete Java programs.
- We present experimental results that evaluate a large segment of the analysis design space.

2 Problem Definition

Our goal is to design a static analysis that answers the following question: given a set of Java classes (i.e., a component to be analyzed), which methods in these classes may produce side effects when invoked from unknown client code written on top of the component? The analysis output is a set of component methods that are guaranteed to be free of side effects when invoked from arbitrary client code.

These analysis results are useful for various users of the component and can be included in the component documentation—for example, Javadoc comments can be added automatically to the source code of the component. This information can be used by programmers that are writing or modifying code built on top of the component, and by software tools that analyze such client code. Having this information is especially important for reusable components (e.g., class libraries) that are developed independently of future clients of the component functionality. Furthermore, the developer of the component can use these results to ensure that the design goals (e.g., `isQuery` attributes) are in fact satisfied by the actual implementation.

The input to the analysis contains a set Cls of interacting Java classes. We will use "classes" to refer to both Java classes and interfaces, since the distinction is irrelevant for this work. A subset of Cls is designated as the set of *accessible classes*; these are classes that may be accessed by unknown client code from outside of Cls. We assume that such client code can only access fields and methods from Cls that are declared in some accessible class.

2.1 Side-effect-free Methods

To define the notion of a side-effect-free method, consider the run-time execution of some client code that is built on top of Cls. Suppose a call site s in the client code invokes a method m defined in Cls. We will refer to such methods as *boundary methods*. An invocation of a boundary method m is free of side effects if the observable state immediately after the completion of the invocation is exactly the same as the observable state immediately before the invocation. The *observable state* at an execution point consists of (1) all static fields, and (2) all heap objects that are transitively reachable from static fields or from locals/formals of methods that are currently on the run-time call stack.

If the invocation of m at s changes the value of some static field, this could potentially affect the client code that is executed after s. Similarly, if the invocation modifies the object structure that is reachable from static fields and from currently-active locals and formals, this may affect the subsequent execution. The change in the object structure could be due to modifications of non-reference fields (e.g., integer fields, boolean fields, etc.). The change could also be due to reference fields: for example, a new object may be created and a reference to it may be assigned to a field of an existing object. Note that side-effect analyses typically consider all changes to the heap object structure to be side effects. On the other hand, our definition takes into account only changes that are observable by the caller, which follows the notion of pure methods used elsewhere (e.g., [13]). A boundary method m from Cls is *side-effect-free* if all possible client invocations of m during all possible run-time executions of arbitrary clients of Cls are free of side effects, subject to the constraints described in Section 2.2.

```
package iter;
public abstract class BreakIter {
   public static BreakIter getWordInstance()
         { return new SimpleBoundary(...); }
   public abstract void setText(CharIter ci);
   public abstract void setText(String s);
   public abstract CharIter getText();
   public abstract int first();
   public abstract int firstPeek();
   public abstract int firstPeek(String s);
   public abstract int next();
   public abstract int nextPeek();  }
final class SimpleBoundary extends BreakIter {
   private CharIter text;
   private int pos;
   public void setText(CharIter ci)
      { text=ci; pos=text.getBeginIndex(); }
   public void setText(String s1)
      { setText(new StringCharIter(s1)); }
   public CharIter getText() { return text; }
   public int first()
      { pos=text.getBeginIndex(); return pos; }
   public int firstPeek()
      { return text.getBeginIndex(); }
   public int firstPeek(String s2) {
       StringCharIter tmp=new StringCharIter(s2);
       return tmp.getBeginIndex(); }
   public int next()
      { pos=nextPosition(pos); return pos; }
   public int nextPeek()
      { return nextPosition(pos); }
   private int nextPosition(int offset)
      { ...  text.nextChar() ...  }  }
public interface CharIter {
   public int getBeginIndex();
   public char nextChar(); ...  }
public class StringCharIter implements CharIter {
   private String txt;
   private int begin=0,curr=0;
   public StringCharIter(String s3) { txt=s3; }
   public int getBeginIndex() { return begin; }
   public char nextChar() {...  curr++; ...} ...}
```

Figure 1. Sample package `iter`.

Example. Consider the package in Figure 1. This example is based on classes from the standard Java library package `java.text`, with some modifications introduced to illustrate aspects of our approach. `BreakIter` allows iteration over different boundaries in text—e.g., boundaries of words, boundaries of sentences, etc. Internally the iteration is implemented by `SimpleBoundary`. The text is accessed through `CharIter`, which defines a protocol for iteration over characters. `StringCharIter` implements this protocol for string objects. Here *Cls* contains the classes from Figure 1 plus `String`. Class `SimpleBoundary` has package visibility and cannot be accessed directly by client code; the remaining classes are designated as accessible classes.

Boundary methods `getText` and `getBeginIndex` are trivially free of side effects. Method `firstPeek()` invokes only `getBeginIndex` and is also side-effect-free. Method `firstPeek(String)` is free of side effects because it creates a temporary object which does not "escape" to the caller of the method, and therefore the caller's observable state does not change. All other boundary methods potentially have side effects.

2.2 Constraints

When considering the effects of a method invocation, it is useful to define certain constraints that allow more precise discrimination of sources of side effects. In our definition of side effects, we employ two such constraints. First, we only consider executions in which the invocation of a boundary method m from *Cls* does not leave *Cls*—i.e., all of m's transitive callees are also in *Cls*. Without this constraint, we would have to assume that unknown called methods always have side effects (e.g., they modify static fields). In particular, if we consider the possibility that unknown subclasses override methods from *Cls*, all instance calls inside *Cls* could potentially be "redirected" to unknown external code, and therefore may have side effects. Thus, the vast majority of methods from *Cls* would have to be reported as having side effects. For example, the call to `getBeginIndex` in `firstPeek()` could potentially have side effects if client code creates and uses some unknown class that implements interface `CharIter`, instead of using `StringCharIter`.

Since the analysis scope is restricted to the given set *Cls*, this set should include all relevant classes whose code is available. For example, in the experiments presented in Section 6, we included in *Cls* all classes that were (transitively) referenced by other classes in *Cls*. With this approach, the user of the analysis is given information that is valid for the currently known set of classes, but may be invalidated in the future by the addition of new subclasses of classes from *Cls*. An alternative approach is to change the analysis to make worst-case assumptions at calls that may leave *Cls* and enter some unknown overriding methods. However, we believe that rather than taking this overly conservative approach, it is more useful to restrict the scope of the analysis to the "known world". Thus, the analysis will produce results that are valid for the particular set of classes that are currently available to the analysis user. Of course, the user must be aware that the definition of "side-effect-free" may not be valid in the context of some larger set of classes which contains new subclasses of classes from *Cls*, and that the analysis will have to be rerun when such subclasses become available.

The second constraint is related to the order of method invocations. We only consider executions in which the invocation of a boundary method $m \in Cls$, and the subsequent invocations of m's callees, are executed in single-

threaded fashion, without interleaving invocations due to other threads. (This approach is traditionally employed by all side-effect analyses.) Otherwise, there is the possibility that an arbitrary method with side effects is executed at any point of time during the execution of any method m, and therefore m cannot be considered to be side-effect-free.

3 Class Analysis

Class analysis determines the classes of all objects to which a reference variable or a reference field may point. This information has a variety of uses in software tools and optimizing compilers. In this paper, class analysis will be used to approximate the calling relationships between methods, as well as the set of objects that constitute the observable state with respect to client code. Our goal is to define a general theoretical framework for analyses that identify side-effect-free methods, based on the design space for class analysis algorithms.

3.1 Categories of Class Analysis

There is a large body of work on class analyses with different cost/precision trade-offs. One dimension of precision is *flow sensitivity*. Flow-insensitive analyses do not take into account the flow of control within a method, and are less expensive and less precise than flow-sensitive analyses. Another dimension is *context sensitivity*: context-sensitive analyses employ some abstraction of the calling context of a method in order to achieve higher precision.

An important analysis aspect is the naming scheme used to distinguish among instances of the same class. In traditional class analyses, no such distinction is made. Other class analyses (also referred to as points-to analyses) create a separate name for each allocation site. In this case, two instances of the same class are modeled differently if and only if they are created by different `new` expressions in the code. To allow uniform treatment of these two approaches, we assume that the analysis uses a set O of *object names* to represent heap objects. In traditional class analysis, there is a single object name per instantiated class or instantiated array type. In points-to analysis, there is a separate object name for each `new` expression.

3.2 Output of Class Analysis

The output of class analysis is usually thought of as a set of relationships of the form "local/formal v may refer to instances of C" and "field f may refer to instances of C". In this work, we assume that the output has a different form that represents exactly the same kinds of relationships, but is more convenient for the subsequent analyses. We consider the output to be a set of *points-to pairs* and a set of *call edges*. Points-to pairs represent "may-refer-to" relationships. Call edges represent possible calling relationships between methods. This definition allows general and uniform treatment for a wide range of flow-insensitive analyses with different degrees of context sensitivity [1, 2, 11, 6, 20, 25, 28, 27, 10, 16, 22, 18, 15].

We will use E to denote the set of call edges. Each call edge is a pair (s, m) where s is a call site and m is a method that is potentially invoked at s. If s is a polymorphic call site, there are multiple call edges (s, m_i). Most class analyses construct E on the fly during the analysis; if this is not the case, it can be constructed after the analysis completes.

Let V be the set of all locals, formals, and static fields that have reference types (class/interface/array types [9]). Set O denotes the set of object names used by the analysis, as described above. Set C contains a set of *contexts*. For context-sensitive analysis, the elements of C represent abstractions of calling context. We assume that C contains a special element ϵ that denotes the "empty" context. For context-insensitive analyses, we can define $C = \{\epsilon\}$. For each call edge $e = (s, m)$, the analysis computes a set of possible contexts $C_e \subseteq C$ that may be introduced by s at the entry of the called method m. Examples of common context abstractions are the top k call sites on the call stack [10], or some $o \in O$ representing the receiver object [18].

The analysis solution represents three categories of "may-refer-to" relationships:

- Consider some $v \in V$ and $o \in O$. Let $c \in C_e$ be a context that is introduced at the entry of some method m by a call edge $e = (s, m)$. If the analysis solution contains a points-to pair (v^c, o), this means that at run time, if m is invoked from s with calling context c, during that invocation v may refer to some object that is represented by o.
- Suppose that f is a reference instance field in objects represented by some $o \in O$. For example, if o represents objects of class C, f can be any reference instance field declared in C or inherited from C's superclasses. The pair $(o.f, o_2)$ shows that at run time the f field of some object represented by o may refer to some object represented by o_2.
- If o represents array objects, $(o[\,], o_2)$ shows that some element of some array represented by o may refer at run time to an object represented by o_2.

4 Fragment Class Analysis

Class analysis is usually designed as *whole-program analysis*: it takes as input a complete program and produces information about relationships in that entire program. However, the problem we are considering requires analysis of partial programs. The input is only the set of

```
void main() {
   BreakIter break_iter;
   CharIter char_iter;
   StringCharIter string_char_iter;
   String string = "string literal";
   break_iter = BreakIter.getWordInstance();
   break_iter.setText(char_iter);
   break_iter.setText(string);
   char_iter = break_iter.getText();
   break_iter.first();
   break_iter.firstPeek();
   break_iter.firstPeek(string);
   break_iter.next();
   break_iter.nextPeek();
   char_iter.getBeginIndex();
   char_iter.nextChar();
   string_char_iter = new StringCharIter(string);
   char_iter = string_char_iter;
}
```

Figure 2. Placeholder `main` method for `iter`.

classes Cls, and the analysis has to take into account possible effects of client code that may be built later on top of Cls. To address this issue, we use an adaptation of earlier work on fragment class analysis [23]. Such analysis works on a program fragment rather than on a complete program; for our work, the fragment is the given set of classes Cls.

The approach produces an artificial `main` method that "simulates" the possible flow of object references between Cls and the client code that may be written on top of Cls. Intuitively, `main` serves as a placeholder of possible client code. The fragment analysis attaches this method to Cls and invokes the engine of some whole-program class analysis. This technique is applicable to many flow-insensitive whole-program class analyses [1, 2, 11, 6, 20, 25, 28, 27, 10, 16, 22, 18, 15]. A wide spectrum of fragment analyses can be defined in this manner: from very simple and inexpensive analyses based on Rapid Type Analysis (RTA) [2], to relatively complicated context-sensitive analyses.

The placeholder `main` method for the classes from Figure 1 is shown in Figure 2. The method contains variables that correspond to different types from Cls that may be accessed by client code. The statements represent different possible interactions involving Cls. The order of statements is irrelevant because the subsequent whole-program class analysis is flow-insensitive. Method `main` invokes all public methods from the classes in Cls which have been designated as accessible classes. The variables in these calls are based on the types in method signatures. The last statement takes into account the possibility of assignment conversions [9] between `CharIter` and `StringCharIter`.

4.1 Fragment Analysis Solution

We will not discuss all details of the approach for creating the placeholder `main` method; additional information is available in [23]. However, it is important to define precisely the properties of the information produced by the fragment analysis, because this information will be used subsequently to identify side-effect-free methods.

Call Graph. Consider some arbitrary client program built on top of Cls, and some execution of this program that satisfies the constraints described in Section 2. For each relationship of the form "during the execution, call site s in method n invokes method m", the fragment analysis solution contains a call edge that represents this relationship. If both n and m are methods in Cls, the solution contains the call edge (s, m). If n is not in Cls, the call graph contains at least one call edge (s, m), where s is some call site inside the placeholder `main` method. Essentially, the call graph computed by the fragment analysis can be thought of as a projection of all possible calling relationships within Cls and between Cls and client code.

Object Reachability. Consider again some client program built on top of Cls, and an execution of this program that satisfies the constraints from Section 2. Suppose $v \in V$ is declared in Cls, and at some point during the execution v is the start of a chain of object references that leads to some heap object. In the fragment analysis solution, there will be a chain of points-to pairs that starts at v^c and leads to some object name o that represents the run-time object. Here $c \in C$ is some context that represents the run-time call stack at the execution point. A similar property holds if v is declared outside of Cls. In this case, in the fragment analysis solution, the starting point of the chain is the variable from `main` that has the same type as v.

4.2 Rapid Type Analysis

We will illustrate these properties for two fragment analyses. The first one is based on Rapid Type Analysis (RTA) [2]. RTA is a simple whole-program analysis that computes a set of methods reachable from `main`, and a set of classes instantiated in reachable methods. It does not distinguish among allocation sites (i.e., there is one object name per instantiated type), and is context-insensitive, with $C = \{\epsilon\}$.

Suppose that the `main` method from Figure 2 is added to the classes from Figure 1 and the result is analyzed by RTA as if it were a complete program. Since `getWordInstance` is called from `main`, it is added to the set of reachable methods. Inside this method, `SimpleBoundary` is instantiated. Thus, the calls through `break_iter` may invoke the corresponding methods in `SimpleBoundary`. Analysis of `main` shows that `StringCharIter` may also be instantiated. In the final solution, all methods in Figure 1 are determined to be reachable, and the instantiated classes are `SimpleBoundary` and `StringCharIter`.

(break_iter,SimpleBoundary)
(char_iter,StringCharIter)
(string_char_iter,StringCharIter)
(ci,StringCharIter)
(tmp,StringCharIter)
(string,String) (s1,String) (s2,String) (s3,String)
(SimpleBoundary.text,StringCharIter)
(StringCharIter.txt,String)

Figure 3. Points-to pairs computed by RTA.

The computed points-to pairs, shown in Figure 3, are based on object names for each instantiated class. For brevity, we do not show all pairs involving implicit parameters `this`. The solution provides conservative information about object reachability. For example, the two pairs (`break_iter`,`SimpleBoundary`) and (`SimpleBoundary.text`,`StringCharIter`) imply that some reference variable, which is declared in client code with type `BreakIter`, may be the start of a chain of object references that leads to an instance of `StringCharIter`.

4.3 Context-sensitive Points-to Analysis

As an example of an analysis at the other end of the cost/precision spectrum, we will also consider a fragment class analysis based on a context-sensitive whole-program points-to analysis for Java [18]. The analysis is a context-sensitive version of flow- and context-insensitive Andersen-style points-to analysis for Java [25, 16, 22, 15]. Unlike RTA, the analysis creates a separate object name for each `new` expression, and uses a set of contexts to approximate the state of the run-time call stack. The set of contexts C is exactly the same as the set of object names O. A context $o \in O$ represents an invocation of an instance method when the receiver object of the invocation is o. The analysis propagates "may-refer-to" relationships by analyzing individual program statements. For example, if the analysis encounters "$p = q$" under context c, it infers that p^c may refer to any object that q^c may refer to. Theoretically, this approach is substantially more precise that RTA.

Consider the example from Figures 1 and 2. In this case there are five object names. The first three, which will be denoted by `SimpleBoundary1`, `StringCharIter1`, and `String1`, correspond to the allocation sites in the `main` method. Name `StringCharIter2` corresponds to the allocation in `setText(String)`, and `StringCharIter3` corresponds to the allocation in `firstPeek(String)`. The points-to pairs are shown in Figure 4; for simplicity, we do not show all pairs involving `this`. For variables in `main`, the calling context is the empty context ϵ. For locals/formals inside `SimpleBoundary` the context is `SimpleBoundary1`, which represents the receiver for methods in this class. For

(break_iter$^{\epsilon}$,SimpleBoundary1)
(char_iter$^{\epsilon}$,StringCharIter1)
(char_iter$^{\epsilon}$,StringCharIter2)
(string_char_iter$^{\epsilon}$,StringCharIter1)
(string$^{\epsilon}$,String1)
(ci$^{\texttt{SimpleBoundary1}}$,StringCharIter1)
(ci$^{\texttt{SimpleBoundary1}}$,StringCharIter2)
(s1$^{\texttt{SimpleBoundary1}}$,String1)
(s2$^{\texttt{SimpleBoundary1}}$,String1)
(tmp$^{\texttt{SimpleBoundary1}}$,StringCharIter3)
implicit formal `this` in the constructor of `StringCharIter`:
  (this$^{\texttt{StringCharIter1}}$,StringCharIter1)
  (this$^{\texttt{StringCharIter2}}$,StringCharIter2)
  (this$^{\texttt{StringCharIter3}}$,StringCharIter3)
(s3$^{\texttt{StringCharIter1}}$,String1)
(s3$^{\texttt{StringCharIter2}}$,String1)
(s3$^{\texttt{StringCharIter3}}$,String1)
implicit formal `this` in `nextChar`:
  (this$^{\texttt{StringCharIter1}}$,StringCharIter1)
  (this$^{\texttt{StringCharIter2}}$,StringCharIter2)
(SimpleBoundary1.text,StringCharIter1)
(SimpleBoundary1.text,StringCharIter2)
(StringCharIter1.txt,String1)
(StringCharIter2.txt,String1)
(StringCharIter3.txt,String1)

Figure 4. Points-to pairs computed by the points-to analysis.

methods in `StringCharIter`, there are three contexts corresponding to the three names for instances of the class.

The results from Figure 4 can be used to infer more precise object reachability relationships than possible with RTA. For example, no reference chains lead to `StringCharIter3` from variables declared in `main`. Therefore, variables from client code cannot be the start of reference chains that lead to objects that are created in `firstPeek(String)`. This information allows the approach to identify method `firstPeek(String)` as being side-effect-free, which is not possible using the RTA-based solution.

5 Side-effect-free Methods

Whole-program side-effect analysis can be used only to compute information about the behavior of a given complete program. In order to analyze partial programs, we define a form of *fragment side-effect analysis* that is based on the output of a fragment class analysis. Furthermore, unlike traditional side-effect analysis, our approach takes into account only changes that are observable by the callers of a method. For this, the analysis identifies the objects that approximate the observable state with respect to client code. Next, it determines which methods may modify fields in such objects. Finally, it performs backward propagation on the call graph to identify methods that have indirect side effects because they invoke methods with side effects.

	Contexts for RTA		Contexts for Points-to Analysis	
	Immediate side effects	Transitive side effects	Immediate side effects	Transitive side effects
`getWordInstance`	ϵ	ϵ	ϵ	ϵ
`setText(CharIter)`	ϵ	ϵ	`SimpleBoundary1`	`SimpleBoundary1`
`setText(String)`	ϵ	ϵ	`SimpleBoundary1`	`SimpleBoundary1`
`getText`	—	—	—	—
`first`	ϵ	ϵ	`SimpleBoundary1`	`SimpleBoundary1`
`firstPeek`	—	—	—	—
`firstPeek(String)`	ϵ	ϵ	—	—
`next`	ϵ	ϵ	`SimpleBoundary1`	`SimpleBoundary1`
`nextPeek`	—	ϵ	—	`SimpleBoundary1`
`nextPosition`	—	ϵ	—	`SimpleBoundary1`
`StringCharIter`	ϵ	ϵ	`StringCharIter{1,2}`	`StringCharIter{1,2}`
`getBeginIndex`	—	—	—	—
`nextChar`	ϵ	ϵ	`StringCharIter{1,2}`	`StringCharIter{1,2}`

Table 1. Contexts under which the methods have immediate and transitive side effects.

5.1 Observable State

The invocations of a side-effect-free method m do not affect the state observable by m's callers. The observable state at a program point consists of all static variables, as well as all heap objects that are transitively reachable from (1) static variables and (2) locals/formals of methods that are currently on the run-time call stack. We first identify the object names which represent objects that are part of the observable state for callers of boundary methods. This is done by traversing the points-to pairs produced by the fragment class analysis. The starting point of the traversal are all variables declared in placeholder method `main`. Any object name that is transitively reachable from these variables is included in the observable state; we will refer to such object names as *observable object names*. Consider the solutions in Figures 3 and 4. For the first solution, the observable state contains object names `SimpleBoundary`, `StringCharIter`, and `String`. For the second one, it contains `SimpleBoundary1`, `StringCharIter1`, `StringCharIter2`, and `String1`.

The observable state contains two kinds of objects: objects that already exist before a call, and objects that are created during the call and "escape" to the caller. Existing work on escape analysis [5, 3, 4, 29] can be used to identify escaping objects. However, we chose to identify escaping objects using fragment class analysis. This is a general approach, it is easy to implement, and allows cost/precision trade-offs through the underlying class analysis. Our experiments indicate that even simple and inexpensive analyses (e.g., based on RTA) can achieve very high precision, and therefore the use of the more expensive escape analyses may be unnecessary for the problem considered in this paper.

5.2 Immediate Side Effects

The second step of the analysis identifies method invocations that have "immediate" side effects. Such invocations either modify the value of a static variable, or modify observable objects. Let $e = (s, m)$ be a call edge and $c \in C_e$ be a calling context for m. The pair (m, c) represents an invocation of m under context c. This invocation has *immediate side effects* if:

- m assigns a value to a static field
- m contains an object allocation site that corresponds to an observable object name
- m assigns a value to `x.f`, where $\texttt{x}^c$ refers to an observable object name and `f` is an instance field
- m assigns a value to `x[i]`, where $\texttt{x}^c$ refers to an observable object name that represents an array object

Table 1 show the contexts c for which a pair (m, c) has immediate side effects. Consider the RTA-based solution; in this case, the analysis uses a single context ϵ. `getWordInstance` creates objects that are represented by the observable object name `SimpleBoundary`, and therefore the method has immediate side effects. Method `setText(CharIter)` contains two assignments to fields: "`this.text=...`" and "`this.pos=...`". The implicit formal `this` refers to the observable object name `SimpleBoundary`, and therefore the method has immediate side effects. Two other methods in the class modify an instance field of an observable object (`first` and `next`), and two methods contain allocation sites which correspond to observable object names (`setText(String)` and `firstPeek(String)`). In class `StringCharIter`, the constructor and method `nextChar` also have immediate side effects. (Alternatively, the constructor could be considered to be free of side effects because it modifies only fields of the newly-created object [13]).

Consider the solution computed using the context-sensitive analysis. `getWordInstance` has immediate side effects because it creates objects that are represented by the observable object name `SimpleBoundary1`. Since this is a static method (i.e., it does not have a receiver object), the analysis uses invocation context ϵ.

(1) Component	(2) Functionality	(3) #Classes	(4) #Boundary Methods	#Side-effect-free methods		
				(5) RTA	(6) ContextSens	(7) Perfect
`gzip`	GZIP IO streams	199	12	0 (0%)	0 (0%)	0 (0%)
`zip`	ZIP IO streams	194	33	9 (27%)	10 (30%)	10 (30%)
`checked`	IO streams with checksums	189	9	2 (22%)	2 (22%)	2 (22%)
`collator`	text collation	203	26	10 (39%)	10 (39%)	10 (39%)
`date`	date formatting	205	23	4 (17%)	4 (17%)	4 (17%)
`number`	number formatting	198	26	6 (23%)	6 (23%)	6 (23%)
`boundary`	iteration over boundaries in text	199	35	8 (23%)	8 (23%)	8 (23%)

Table 2. Java components and side-effect-free methods.

Methods `setText(CharIter)`, `first`, and `next` modify instance fields of the observable object name `SimpleBoundary1`. Method `setText(String)` creates objects that are represented by the observable name `StringCharIter2`. In `firstPeek(String)`, the allocation corresponds to `StringCharIter3` which is not part of the observable state. Therefore, the method does not have immediate side effects.

Two methods from `StringCharIter` have immediate side effects for calling contexts `StringCharIter1` and `StringCharIter2`. Name `StringCharIter3` is also a valid context for the constructor; however, under this context there are no modifications of observable objects.

5.3 Transitive Side Effects

The final step of the analysis considers indirect side effects that are due to called methods. Consider an invocation represented by a pair (m, c) of a method and a calling context. The invocation has *transitive side effects* if at least one of the following holds:

- (m, c) has immediate side effects
- m contains a call site s such that when m is invoked under c, site s invokes a method m' with context c' and the invocation (m', c') has transitive side effects.

Table 1 shows the pairs (m, c) with transitive side effects. `getText` and `getBeginIndex` do not have such side effects. `firstPeek` only invokes `getBeginIndex` and does not have transitive side effects. `nextPosition` and `nextPeek` have transitive side effects because they invoke `nextChar`. In the context-sensitive solution, the constructor invocation in `firstPeek(String)` creates calling context `StringCharIter3`; under this context, the constructor does not have immediate side effects, and the caller does not have transitive side effects.

Once the analysis computes all pairs (m, c) that have transitive side effects, it considers the call sites inside the `main` method. If such a call site invokes some boundary method m from Cls under some context c, and if (m, c) has transitive side effects, the method is reported as potentially having side effects that may be observed by client code. All remaining method are guaranteed to be free of side effects when invoked by client code. The RTA-based approach identifies the side-effect-free methods `getText`, `firstPeek`, and `getBeginIndex`. Using the context-sensitive analysis, method `firstPeek(String)` can also be identified as having no side effects.

6 Experimental Study

The goal of this study is to address two questions. First, how often do different analyses discover side-effect-free boundary methods? Second, how significant is the *imprecision* of the analyses—that is, how often do they miss methods that in fact are free of side effects? The answers to these questions provide essential insights for analysis designers and tool builders.

For the first question, we performed experiments with two class analyses at the opposite ends of the cost/precision spectrum: RTA and the context-sensitive points-to analysis described in Section 4. For the experiments we used several Java components from the standard library packages `java.text` and `java.util.zip`. The components are described briefly in the first two columns of Table 2. Each component contains a set of classes that provide certain functionality; as discussed in Section 2.2, set Cls contains these classes plus all other classes that are directly or transitively referenced by them. Column (3) shows the size of Cls. We then considered all boundary methods for the particular component functionality; the number of such methods is given in column (4).

6.1 Number of Side-effect-free Methods

In order to determine which boundary methods were side-effect-free, we applied the approach presented earlier, using as basis RTA and the points-to analysis. Columns (5) and (6) show how many of the methods from column (4) were identified as being free of side effects.

On average, the analyses reported 22% side-effect-free methods. These results indicate that a large number of methods are free of side effects, and that the analyses

can compute useful information for various software tools. Somewhat surprisingly, the two versions of the analysis achieve essentially the same precision. The only difference is in `zip`, where the theoretically more precise analysis identifies an additional method. Since the difference between the two analyses covers a large portion of the analysis design space, these results indicate that very precise identification of side-effect-free methods is possible with simple and inexpensive analysis techniques at the low end of the cost/precision spectrum. These observations are promising because they suggest that such functionality can be easily incorporated in any software tool.

6.2 Analysis Imprecision

The issue of analysis imprecision is of critical importance for software tools. If an analysis is imprecise, it may report that methods have side effects even if in reality they do not. Such information is of little use (or even confusing) for tool users. For example, if a user attempts to ensure the consistency between the code and the `isQuery` properties in a UML-based design, imprecision would mean that the code of a method (and all its transitive callees) will have to be inspected manually to identify the source of the discrepancy. Because of the negative effects of such imprecision, it must be evaluated carefully and precisely. Unfortunately, the traditional approach for precision evaluation used in static analysis research does not provide sufficient information to evaluate this imprecision [21]. Usually, precision evaluations compare the results of two or more static analyses against each other. However, this by itself does not indicate how far away the analyses are from the "perfect" solution which does not contain any imprecision.

In our experiments, we carefully examined all boundary methods that were reported as having side effects, and we attempted to prove that it was possible to write client code which observed such side effects. More precisely, we attempted to prove that there exists a run-time execution of some client code during which a method invocation has side effects according to the definition from Section 2. Essentially, these were proofs by existence. The results are shown in column (7) in Table 2. In all cases, we were able to prove that the methods reported by the points-to analysis as having side effects were not side-effect-free in reality. Therefore, this analysis achieves *perfect precision*. The RTA-based analysis achieves almost perfect precision, missing only one method that is actually side-effect-free.

6.3 Conclusions

The results presented above are promising because they indicate the presence of a large number of side-effect-free methods. Thus, the analysis can provide useful information for a variety of software tools. Furthermore, there is strong indication that highly precise information can be obtained with techniques that are inexpensive and simple to implement. Of course, these results need to be reconfirmed for more components. At this point we can draw two conclusions. First, any future investigations should focus, at least initially, on analyses at the low end of the cost/precision spectrum. Second, the precision of the analysis results should be evaluated not by comparing them with other analyses, but directly with the "perfect" solution.

7 Related Work

There is a large body of work on interprocedural side-effect analysis. For languages with general-purpose pointers (e.g., C), such analysis must be preceded by a pointer analysis that disambiguates memory accesses. Ryder et al. [24] present a general framework for side-effect analysis for C programs, parameterized by pointer analysis. If the pointer analysis is context-sensitive, the propagation of side effects is also done in context-sensitive fashion. Our approach is based on a similar idea: given a context-sensitive class analysis, we take into account different calling contexts when considering whether a method has immediate or transitive side effects. However, our approach is designed to work on partial programs, and to consider the observable state with respect to client code; the schema from [24] is for whole-program analysis. Furthermore, we directly evaluate the degree of imprecision in the analysis solution.

Razafimahefa [20] presents algorithms for side-effect analysis for Java that are based on context-insensitive class analyses. Milanova et al. [18] propose a side-effect analysis for Java based on the particular form of context-sensitive class analysis described in Section 4. In both cases, the side-effect analysis is designed as a whole-program analysis. Our approach is more general because it allows analysis of partial programs, and is parameterized by a wide range of class analyses (both context-sensitive and context-insensitive). We also present an evaluation of the analysis imprecision, which is essential information for designers of software tools.

Recent work by Sălcianu and Rinard [26] presents a static analysis which identifies pure methods in Java code. The analysis also produces additional information—for example, regular expressions describing the heap objects that are modified by a method with side effects. Their technique is based on the pointer and escape analysis from [29]. Our work takes a different approach: instead of using a specialized pointer/escape analysis, we use the points-to relationships and calling relationships computed by a fragment class analysis to identify the observable objects that may be modified. This allows the use of a variety of existing class analyses [1, 2, 11, 6, 20, 25, 28, 27, 10, 16, 22, 18, 15].

8 Conclusions and Future Work

This work defines an approach for performing analysis of side-effect-free methods in Java components. Our study indicates that a large number of methods can be identified as having no side effects, and typically *all* side-effect-free methods can be successfully identified. Clearly, it would be premature to draw far-reaching conclusions from these experiments. However, as a first step in investigating this problem, the results are promising. Furthermore, they provide a clear direction for future investigations. Through additional experiments by us and by other researchers, it may become possible to conclude that precise identification of side-effect-free methods can be done easily and efficiently. We plan to perform such investigations in the future.

References

[1] O. Agesen. The cartesian product algorithm. In *European Conference on Object-oriented Programming*, LNCS 952, pages 2–26, 1995.

[2] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[3] B. Blanchet. Escape analysis for object-oriented languages. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 20–34, 1999.

[4] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 35–46, 1999.

[5] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[6] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[7] J. J. Dolado, M. Harman, and M. C. Otero. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Trans. Software Engineering*, 29(7):665–670, July 2003.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2 edition, 2000.

[10] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[11] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, 1997.

[12] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2nd edition, 2002.

[13] G. Leavens, A. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[14] R. Lencevicius, U. Hölzle, and A. Singh. Query-based debugging of object-oriented programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 304–317, 1997.

[15] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[16] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, 2001.

[17] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *International Conference on the Unified Modeling Language*, LNCS 2460, pages 426–441, 2002.

[18] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[19] OMG. *UML 2.0 Infrastructure Specification*. Object Management Group, `www.omg.org`, Aug. 2003.

[20] C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, Dec. 1999.

[21] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Workshop on Program Analysis for Software Tools and Engineering*, June 2004.

[22] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, Oct. 2001.

[23] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Trans. Software Engineering*, 30(6):372–387, June 2004.

[24] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Programming Languages and Systems*, 23(2):105–186, Mar. 2001.

[25] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, U. Passau, Sept. 2000.

[26] A. Sălcianu and M. Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, MIT, May 2004.

[27] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[28] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.

[29] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.