Static vs. Dynamic Purity Analysis

Valentin Dallmeier
Dept. of Computer Science
Saarland University
Saarbrücken, Germany
dallmeier@cs.uni-sb.de

ABSTRACT

A method is *pure* if its invocation does not cause externally visible side-effects. We describe a new approach to dynamic purity analysis based on the observation of side-effects at runtime and compare our results to a state-of-the art static purity analysis. In our experiments with a large open-source project, 35% of the methods statically classified as impure never show side-effects in more than 600 regression tests. We use the results of our analysis as input to ADABU, a dynamic tool for mining object behavior models. Our experiences show that dynamic analyses do not require static purity classification but also work well with the results of a dynamic analysis. Our tool JPURE is publicly available so that other researchers can benefit from our work.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects; Procedures, functions, and subroutines*

1. INTRODUCTION

In object-oriented languages, the execution of a method can have side-effects, i.e. the method modifies an externally visible object or the global state (the values of all static fields). A *pure* method is a method that never has side-effects, whereas an *impure* method may have side-effects. As an example consider the two methods in Figure 1: n is impure because it modifies v and w. m however is pure, because it only modifies objects that are not visible externally.

A purity analysis classifies methods as either pure or impure. This classification is used in many different areas: Several program analyses benefit from knowing if the execution of a method may change the visible state [10, 15]. It is also useful for program understanding and documentation [8], verification [4] and specification [3]. Previous work has employed static [17, 18, 19] and dynamic [22] techniques to analyze method purity.

Parameter mutability analysis is closely related to purity analysis. It classifies all complex (i.e. non-primitive) parameters of a method as either *immutable* or *mutable*. In the example in Figure 1,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

```
1  public void m(Object o) {
2    Vector v = n(new Vector());
3    v.add(2);
4  }
5  public Vector n(Vector v) {
6    v.add(1);
7    return new Vector();
8 }
```

Figure 1: Method n is impure because it modifies v. m is pure because all modified objects are created during the invocation of m.

o is immutable and v is mutable due to the modification in line 6. Mutability information is useful for dynamic detection of invariants [9], program slicing [8] and program comprehension [21]. Previous work [1] has employed both static and dynamic techniques to analyze parameter mutability.

This paper presents a new approach to dynamic purity analysis based on the observation of side-effects at runtime. The approach is simple and can easily be extended to analyze parameter mutability. Our work is the first to evaluate the precision of a dynamic purity analysis by comparing the results to a state-of-the-art static purity analysis. We make the following contributions:

- The description of a simple and effective way to compute dynamic purity information following a well-established purity criterion [19] (Section 2). With small adjustments, our approach is also able to compute object immutability information for parameters (Section 3).

- An evaluation of the soundness of both analyses. To the best of our knowledge, our work is the first to provide an evaluation of the precision (how many methods were classified as pure although they may have side-effects) of a dynamic purity analysis (Section 4). Our results indicate that 35% of the methods statically shown to be impure never have side-effects at runtime.

- An implementation of our approach in a tool called JPURE. The tool is capable of analyzing programs as large as ECLIPSE with acceptable overhead. JPURE is publicly available so that other researchers can benefit from our work.

- We discuss our experiences with ADABU (Section 5), a tool that invokes side–effect free methods to for state inspection. By using JPURE as purity analysis, ADABU was able to analyze programs as large as ECLIPSE without errors, suggesting that dynamic purity information is sufficient for dynamic analysis techniques.

2. PURITY ANALYSIS

Our approach uses the same purity criterion as [19]: A method *m* is pure if it never modifies an object that existed prior to it's invocation. Thus, *m* can create and modify temporary objects and invoke impure methods as long as the side-effects of these invocations only alter objects created during the invocation of *m*. This is a more flexible criterion than the one used by previous analyses such as [4] that disallow any modifications to the heap. An example that illustrates the purity criterion is depicted in Figure 1.

The idea behind the analysis is as follows: For each invocation of a method *m*, we record the IDs of all objects created during that invocation. This information is used to calculate a *set of allocated objects* for each method invocation. In addition, we capture all field write operations and trace the ID of the object that was changed by this operation (static field writes are recorded as changing a special object that represents all static variables). This information is used to calculate the *set of modified objects* for each method invocation. At the end of each invocation, we compare these two sets. Whenever an invocation of a method modifies an object that was not created during that execution, we know that the invocation changed externally visible state. Our analysis classifies a method *m* as pure if it was executed at least once and no execution of *m* modifies externally visible state. If at least one execution of *m* modifies externally visible state, *m* is classified as impure. The output of our analysis consists of a set of pure and a set of impure methods.

We have implemented our analysis for Java programs in a tool called JPURE. The tool works in two phases: In the first phase, one or more executions of the program to be analyzed is monitored (see Section 2.1) and trace data is written to a file. In the second phase (explained in Section 2.2) the trace files are analyzed and purity is calculated.

2.1 Tracing

This section explains how we record the information needed for the analysis. JPURE uses the ASM [11] instrumentation framework to rewrite the byte code of the subject program. Table 1 lists all events handled by JPURE and the information that is recorded for each event. In addition to that, our tool handles some special cases as follows:

Multithreading If more than one thread is active at the same time, events triggered by different threads may spoil the results of the purity analysis. To avoid this, every event records an identifier for the thread that triggered the event.

Method Start/End We record the begin and end of each method invocation to allow the analysis to determine the currently active method. We instrument the beginning of each method (except constructors) to record a event. Method end events are recorded before every return instruction. In addition, every method is surrounded by a try-finally block to trace exceptional method exits.

Constructors Every constructor modifies fields of the object it is invoked on. Technically, an object is created before the invocation of the constructor, which is why our analysis would classify every constructor as impure. To avoid this, we add an additional object creation event with the ID of the *this* object.

Arrays Single-dimensional arrays of non-primitive types are treated the same way as fields. Multi-dimensional arrays are processed recursively: For each dimension of the array, we trace $OBJECT_NEW$ events for all values in this dimension.

Events are written sequentially into a single trace file. In order to increase the responsiveness of the application, we use a producer-consumer buffer and a dedicated thread to write data to the file. Upon shutdown of the virtual machine, buffered data is flushed to the disk.

Event	Traced Data	Instrumentation Sites
$METHOD_START$	ThreadID, MethodID	Start of each method
$METHOD_END$	ThreadID, MethodID	ARETURN, DRETURN, FRETURN, IRETURN, LRETURN, RETURN
$OBJECT_NEW$	ThreadID, ObjectID	NEW, NEWARRAY, ANEWARRAY, MULTIANEWARRAY
$FIELD_WRITE$	ThreadID, ObjectID	PUTFIELD, PUTSTATIC, AASTORE, BASTORE, CASTORE, DASTORE, FASTORE, IASTORE, SASTORE

Table 1: Events traced by JPURE.

2.2 Analysis

In the second phase, the trace file is analyzed to compute the set of pure methods. Algorithm 1 shows the key elements of the computation. For each method invocation we manage a set of objects created during the invocation (newObjects) and a set of objects modified during the invocation (modifiedObjects). At the end of each invocation we check for objects that were modified but not created during that invocation (lines 14–15). If at least one such object is found, the method is marked as impure. Otherwise, the method is marked as pure unless it was marked as impure in previous invocations. The last step at the end of each invocation (lines 20 and 21) propagates the sets of new and modified objects into the corresponding sets for the calling method.

For a given invocation of method m, algorithm 1 computes method purity according to the criterion described in Section 2. Side–effects due to field writes in m itself are detected because we trace all field writes and instantiations of objects. Side-effects that occur in methods called by m are detected because the analysis propagates the sets of new and modified objects from all methods invoked by m. The algorithm also allows for m to call an impure method n (cf. Figure 1), as long as all externally visible side-effects of n modify only objects created during the invocation of m.

Our algorithm also handles field writes to transitively reachable objects correctly. For example, suppose that a method m changes field $o.x.f$ where o was not created by m (and thus the side-effect is externally visible), and x currently points to object l. If l was not created by m, l is not in the set of objects created by m and thus the algorithm correctly classifies m as impure. If l was created by m, there must have been a field write that set $o.x = l$ before. Our algorithm classifis this field write as externally visible side-effect since o was not created by m.

2.3 Multiple Program Runs

JPURE allows a user to provide more than one run as input to the purity analysis. We use this feature in the evaluation in Section 4 to analyze how much the soundness and completeness of our analysis benefits from additional data. If multiple runs are available, our tool analyzes them successively and updates the classification as follows:

Algorithm 1 Compute purity information

Input: Trace File f
Output: List of impure and pure methods

```
 1: procedure PURITY(File f)
 2:     /* Initialize Datastructures */
 3:     for all event e ∈ f do
 4:         if e == METHOD_START then
 5:             newObjects.push(new Set());
 6:             modifiedObjects.push(new Set());
 7:         else if e == OBJECT_NEW then
 8:             newObjects.peek().add(objectId);
 9:         else if e == FIELD_WRITE then
10:             modifiedObjects.peek().add(objectId);
11:         else if e == METHOD_END then
12:             Set mObjects = modifiedObjects.pop();
13:             Set nObjects = newObjects.pop();
14:             Set escapes = mObjects.minus(nObjects);
15:             if escapes.isEmpty() then
16:                 /* mark method as pure unless it*/
17:                 /* was marked impure before*/
18:             else
19:                 /* mark method as impure */
20:             end if
21:             modifiedObjects.peek().addAll(mObjects);
22:             newObjects.peek().addAll(nObjects);
23:         end if
24:     end for
25:     emit results;
26: end procedure
```

- If a method m was not analyzed before, we add m to the set of classified methods.
- If m was analyzed before and the new classification is pure, the classification of m remains unchanged.
- If m was analyzed before and the new classification is impure, the classification is set to impure.

2.4 Soundness and Completeness

Deciding wether or not a method has side-effects in general is undecidable. Thus, no analysis can be both *complete* (classifies all methods) and *sound* (all classifications are correct). Since our analysis is a dynamic technique, completeness depends on the amount of code covered by the execution. Purity analysis is a binary classification, so we distinguish two different types of soundness: An analysis is *p-sound* if all methods classified as pure are indeed pure. Conversely, an analysis is *i-sound* if all methods classified as impure are indeed impure. Our analysis is i-sound, because it classifies only those methods as impure that were experimentally shown to have side-effects. However, our technique is potentially not p-sound, since a method may contain code blocks with side-effects that are not executed during the traced execution. The evaluation in Section 4 measures the degree to which our analysis is not p-sound. The type of soundness required depends on the application that uses the results. Code optimization techniques ([16] typically require p-sound techniques. In Section 5, we discuss an application that actually requires p-soundness but works well with the results provided by our analysis.

```
1 public void m(Vector u, Vector v) {
2   u.add(1);
3   Vector w = v;
4   w.add(1);
5 }
```

Figure 2: Parameter u is reference mutable, v is reference immutable and object mutable.

3. PARAMETER MUTABILITY ANALYSIS

For the remainder of the paper, we will use the term parameter instead of complex parameter, since mutability analysis only classifies complex parameters.

Parameter mutability analysis classifies each parameter of a method as either *mutable* or *immutable*. There are two types of immutability: *Reference immutability* guarantees that a given reference is not used to modify an object, whereas *object immutability* ensures that an object passed as a parameter is not modified by a method. In the example in Figure 2, u is reference mutable due to line 2. v is reference immutable, since the parameter reference is not used to mutate v. However, v is object mutable: In line 3, v is aliased to w and line 4 changes w.

In this section we show how to extend the purity analysis presented earlier to compute object immutability for all parameters of a method. Our approach is similar to the alternative dynamic analysis presented in [1]. However, while [1] computes reference immutability, our tool computes object immutability.

In order to calculate parameter mutability, JPURE traces additional information for the following two events (cf. Section 2.1):

- $METHOD_START$ At the beginning of each method invocation, we also trace the IDs of all parameters.
- $FIELD_WRITE$ For every field write to a complex field, we trace the address of the object the field points to after the write.

With this data available, we can extend the algorithm presented in Section 2.2 as follows:

- We maintain a dynamic model of the heap in the following way: For every object o created during the program run, and every complex field $o.f$, we trace the identifier of the object $o.f$ points to. Whenever a field write changes the value of $o.f$, the dynamic heap model is updated.
- At the end of an invocation of method m, our analysis performs the following steps for each parameter p of m:
  - We traverse the dynamic heap model to calculate the set of objects transitively reachable from p.
  - If p itself or any of the objects transitively reachable from p were modified during the invocation of m, p is marked as mutable.
- For each method m that was executed at least once, we classify as immutable all parameters that were not marked as mutable.

Parameter mutability analysis in general is undecidable. Following the terminology by [1], a mutability analysis is *i-sound* if all parameters classified as mutable are indeed mutable and an analysis is *m-sound* if all parameters classified as immutable are indeed

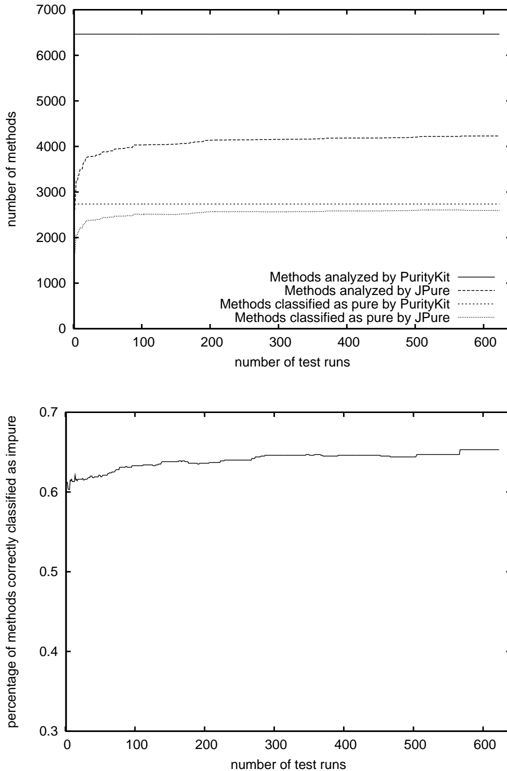

Figure 3: Purity results for ASPECTJ: Coverage increases from 29% at the beginning to 65% with all runs (upper graph). The percentage of methods correctly classified as pure increases as more runs are included (lower graph).

immutable. The analysis presented in this section is m-sound since a parameter as only classified mutable if it has been experimentally shown to be mutable. However, the analysis is not *i-sound* since statically possible modifications of a parameter may not be executed by the runs available.

The analysis handles field writes to transitively reachable objects correctly because we process the dynamic heap model only at the end of an invocation (cf. Section 2.2).

4. EVALUATION

This section presents the evaluation of our tool JPURE. Our analysis uses the same purity criterion as described in [19]. The authors of [19] have implemented their approach in a tool called PURITYKIT, which is publicly available. We use the output of this tool as a ground truth for the purity analysis. Unfortunately, we cannot do the same for the mutability analysis, since JPURE computes object immutability while PURITYKIT computes reference immutability. Nevertheless, we compared mutability results to study the differences between the two types of immutability. Our experiments investigate the following questions:

Soundness How unsound is the purity analysis, i.e. how many entities are classified incorrectly?

Different Types of Mutability How big is the difference between object and parameter mutability results?

Multiple Runs How do completeness and soundness improve when more than one run of a program is used?

Runtime Overhead and Analysis Time How big is the runtime overhead imposed by collecting trace information and how long does it take to process the trace files?

4.1 Measuring Soundness

To measure the soundness of the purity analysis, we calculate the precision values for correct classifications into pure and impure methods (similar to [1]):

$$\text{imp-prec} = \frac{ii}{ii+ip} \quad \text{pure-prec} = \frac{pp}{pp+pi}$$

ii is the number of methods correctly classified as impure, ip is the number of impure methods incorrectly classified as pure, pp is the number of methods correctly classified as pure and pi is the number of pure methods incorrectly classified as impure. imp-prec measures the percentage of methods correctly classified as impure, and pure-prec measures the percentage of methods correctly classified as pure. Ideally, both precision values should be 1.0.

For all experiments with the purity analysis, pi was always equal to zero and thus pure-prec $= 1$. This is due to the fact that all entities classified as impure by our analysis were experimentally shown to be impure (cf. Section 2). On the other hand, our analysis misclassifies impure methods as pure if side-effects are statically possible but never occur at runtime. The discussion in Section 4.3 therefore uses imp-prec to measure soundness.

4.2 Evaluation Subjects

The following subjects were used in our experiments:

JOLDEN [13] is a suite of 10 computationally intensive benchmark programs, which we included because [19] also studies them. We did not include the *power* benchmark in our evaluation since our tool ran out of memory when performing the analysis. *power* creates a large number of temporary objects and our tool tracks data for all of them, even after they are deleted, which eventually exhausts main memory. One way to improve on this would be to trace deletion of objects.

ASPECTJ [7] is a large program with $75k$ lines of code which we used as the main subject to evaluate soundness and completeness of our tool. We study version #29934 of ASPECTJ as provided by the iBUGS [7] repository. We analyzed all 630 test runs available in the regression test suite that comes with the program. In order for PURITYKIT to be able to fully analyze ASPECTJ, we changed one line in the main class to directly instantiate a type rather than using reflection to do so.

ECLIPSE [14] is a large IDE for JAVA programs which we included to show how our tool performs on large interactive programs. We collected data from a run where we created a new class, typed in the body of a main method, compiled and executed the program and closed the workbench. Section 4.5 describes the runtime impact of our tool on the execution time.

The following sections discuss the results for the purity analysis (Section 4.3) and the mutability analysis (Section 4.4). The runtime overhead of our method is discussed in Section 4.5.

4.3 Purity Analysis Results

We used the JOLDEN programs and ASPECTJ as subjects for our evaluation. The graphs in this section present results for ASPECTJ. The precision values for JOLDEN were slightly better than those for ASPECTJ, but coverage was much higher. We focus on ASPECTJ because it is a far more realistic subject than the JOLDEN programs. As mentioned in Section 2.3, our tool can analyze data from more than one run. We take advantage of this feature to perform a cumulative analysis of all tests in the test suite of ASPECTJ. The X-axis of all graphs in this section is the number of runs used by the analysis.

The results for the purity analysis are depicted in Figure 3. The upper graph compares the number of methods analyzed by PURITYKIT and JPURE. When run on ASPECTJ, PURITYKIT classifies a total of 6464 methods. The coverage values for JPURE range from 29% (one run) to 65% (all runs), meaning that JPURE classifies roughly two thirds of the methods classified by PURITYKIT. The precision evaluation uses only methods classified by both tools.

The lower graph in Figure 3 shows the effect of including more runs on the precision of the analysis. With data from one run, JPURE achieves a precision of 0.61. If all runs are included in the analysis, precision is 0.65, which means that our analysis incorrectly classifies 35% of the methods statically shown to be impure as pure. We investigated a sample of these methods. In many cases, the mutating statements are either part of an if-else construct or belong to a method that is the possible target of a dynamically bound call. We also investigated how many times the classification of a method changes from pure to impure when adding data from another run. We found 171 methods where this was the case, which is a small number compared to the total number of classified methods (4231).

To summarize, 35% of the methods classified as impure by the static analysis show no side-effects at runtime. Using more than one run has a positive effect on the precision of our analysis, but the increase is not very strong (0.61 vs. 0.65).

4.4 Mutability Analysis Results

Coverage of parameters increases from 19% with one run to 42%, which is comparable to the increase in coverage of the purity analysis. As already mentioned, PURITYKIT investigates reference immutability while JPURE reports object immutability. Thus the results are not directly comparable. However, we can use the results of both tools to compare object immutability with reference immutability.

For each parameter p classified by both tools, the following four cases are possible:

1. If both tools classify p as immutable, p is reference immutable but potentially object mutable.

2. If both tools classify p as mutable, p is reference and object mutable.

3. If PURITYKIT classifies p as mutable and JPURE classifies p as immutable, p is reference and object mutable. For ASPECTJ, the precision of the mutability classification (cf. Section 4.1) is 0.54, which is less than the precision for the purity analysis.

4. If PURITYKIT classifies p as immutable and JPURE classifies p as mutable, p is reference immutable but object mutable. For ASPECTJ, we found a total of 48 parameters for which this is the case (cp. Figure 4, lower graph) and a total of 2496 reference mutable parameters. These numbers indicate that in 98% of all cases where a method modifies a parameter, it uses the parameter reference to do so.

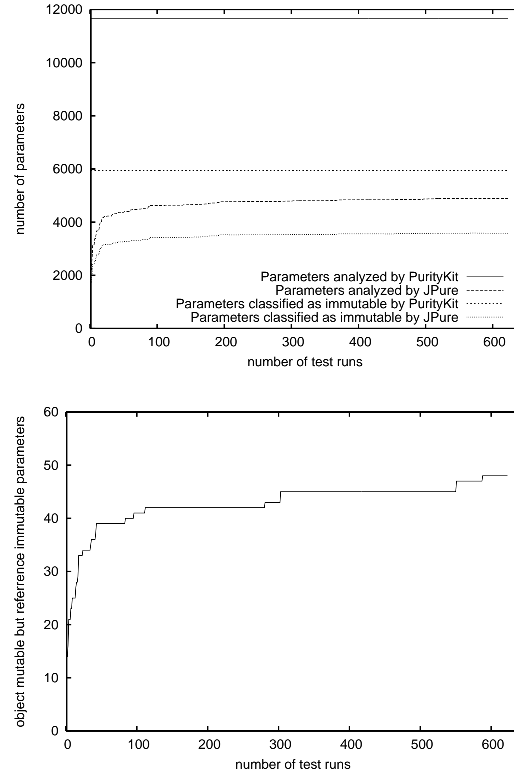

Figure 4: Mutability results for ASPECTJ: 2% of the parameters classified as reference immutable by PURITYKIT are classified as object mutable by JPURE.

To summarize, our results indicate that the difference between reference and object immutability is very small, as only 2% of the paramters that are reference immutable are modified at runtime. Since our analysis is not i-sound, the actual number of such parameters may be higher. However, given that our analysis reaches a precision of 0.54, we expect that less than 4% of the parameters are reference immutable but object mutable.

4.5 Runtime Overhead and Analysis Time

Table 2 lists the results of our overhead evaluation. All experiments were conducted on an AMD64 machine with 2.1Ghz and 2GB of RAM. To capture the runtime overhead we divided execution times (as measured by the unix `time` tool) of a traced run by the execution time of an unmodified run. The resulting values are given in column "Overhead". Columns "Purity Analysis" and "Mutability Analysis" list the number of seconds it took to run the purity analysis and the mutability analysis respectively. The values for ASPECTJ are mean values for all 630 runs of the test suite.

Runtime overhead for the JOLDEN benchmark (except for tsp) is generally much higher than for the other subjects (except for tsp). We attribute this to the fact that the benchmarks are purely memory-based and do not perform I/O-operations. The overhead for the non-benchmark programs ranges from 260 to 420%. The size of a trace file can be up to several gigabytes. Analysis times range from several seconds (treeadd) up to 20 minutes (bh). Muta-

Program	Overhead (factor)	Mutability Analysis Time (seconds)	Purity Analysis Time (seconds)
bh	39.1	1177.38	832.22
em3d	10.2	36.13	33.12
bisort	625.2	619.61	412.92
health	60.2	622.59	155.29
mst	28.8	974.31	602.12
perimeter	41.1	588.94	533.92
treeadd	8.1	771.29	389.72
tsp	3.2	212.05	158.53
voronoi	93.1	760.46	22.25
ASPECTJ	4.3	275.12	86.00
ECLIPSE	5.2	832.46	472.91
COLUMBA	3.6	340.10	182.17

Table 2: Runtime overhead and analysis time for JOLDEN benchmark programs (upper part) and and interactive programs (lower part).

bility analysis generally takes more time since we need to establish a dynamic model of the heap.

To summarize, our tool induces a considerable amount of overhead for the purely memory-based benchmark programs, whereas the overhead for programs that perform I/O is acceptable. Section 7 presents ideas how to reduce the overhead.

4.6 Threats to Validity

Our evaluation only includes results for 10 programs, 9 of which are small benchmark programs. However, due to technical limitations of the tool it is difficult to find subjects that can be analyzed by PURITYKIT. Our main subject ASPECTJ is a large open-source project with several years of history which gives us reason to believe that the results hold for a large number of projects.

Another threat to validity is the use of PURITYKIT as a ground truth. If the results of PURITYKIT were at least partly invalid, this would affect the correctness of our precision results. However, we manually verified a sample of the results and found no misclassifications. Besides that, [1] reports very high precision values for the mutability analysis of PURITYKIT, which gives us reason to believe that the purity analysis achieves similar precision.

5. EXPERIENCES

ADABU [6] is a tool that mines object behavior models from program runs. Such models are finite state automata where states represent the state of an object as captured by the return values of *observer* methods (methods that reveal information about the internal state of an object, such as method `isEmpty()` in class `Vector`). Transitions between states occur whenever a *mutator* method (a method that changes the state of an object such as method `add()` in class `Vector`) is invoked. These methods are called *mutators*. Object behavior models are an elegant means to capture temporal properties of an object's API. Among other applications, they can be useful for program understanding.

ADABU uses the results of a purity analysis to group methods into observers and mutators. Previously, we have used PURITYKIT to provide side-effect information. This severly limited the set of programs we could use ADABU on, since PURITYKIT is not able to analyze programs that require JDK 1.2 or above (which is true for most programs).

To overcome these limitations, we replaced PURITYKIT with JPURE. This requires to execute the subject program at twice: Once to analyze purity with JPURE, and once to actually mine models using ADABU. This is slightly problematic for multithreaded programs, as it is difficult to execute them in exactly the same way more than once. We found this to be no problem in practice, even with large interactive programs: In combination with JPURE, ADABU has successfully mined models from ECLIPSE [14] and Columba [12] (cp. Figure 5). We found no case where the method calls injected by ADABU had a visible impact on the program run suggesting that the purity information provided by JPURE is correct.

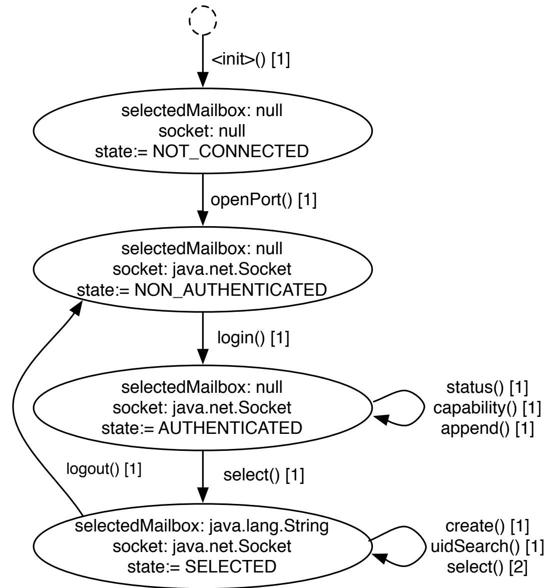

Figure 5: Object behavior model for class `IMAPProtocol`. JPURE is needed to mine this model as existing purity analysis tools cannot analyze the program.

To summarize, JPURE has successfully replaced the static purity analysis we have used before, which greatly improves the applicability of ADABU.

6. RELATED WORK

Side-effect analysis originated more than 30 years ago in the area of compiler construction, where it is used for optimization techniques. Early work by Banning [2] identified the problem of side-effect analysis. Cooper et al. [5] present a flow-insensitive side-effect analysis linear in the size of the call multigraph. These approaches where among the first to identify the core problems and the need for interprocedural analysis. Emerging techniques such as object-oriented languages further complicate the analysis e.g. due to dynamic dispatch. More recent techniques focus on the analysis of Java programs. Milanova et al. [17] use context sensitive points-to information to analyze each method invocation in the context of the object the method is invoked on.

Rountev [18] proposes a static side-effect analysis that can be parameterized by different types of class analysis. The paper compares results achieved with RTA to those for a context sensitive pointer analysis and finds roughly equivalent precision. Their purity criterion used is roughly equivalent to our criterion. However, the technique has difficulties with programs that use reflection, which holds for many modern programs such as ECLIPSE [14].

Xu et al. [22] describe a dynamic purity analysis. Their work explores several purity criteria ranging from strong to weak, whose

definitions are strongly influenced by the proposed application, memoization. In contrast to their work, our analysis explores a well-established purity criterion. They also didn't evaluate the precision of their approach.

Sălcianu et al. [19] have implemented a static purity and parameter mutability analysis in a tool called PURITYKIT. The approach is based on a combined pointer and escape analysis and is the first to allow pure methods to modify objects created during an invocation. The main weakness of their tool is that it cannot analyze programs which require JDK 1.2 or above, while our tool has successfully analyzed modern programs with graphical user interfaces such as ECLIPSE [14].

Artzi et al. [1] propose a combination of static and dynamic techniques for mutability analysis. The different techniques are organized in a pipeline where each analysis incorporates the results from the previous stage. The paper explores various combinations of different analyses to support different types of soundness (cf. Section 3). Their approach computes reference immutability, while we analyze object immutability.

7. CONCLUSIONS

We have presented an approach for dynamic purity analysis that is based on the observation of mutations at runtime. The approach is simple and can easily be extended to compute parameter mutability information. We have implemented our approach in a tool called JPURE. The implementation is stable and has been used to analyze large interactive programs.

To measure the precision of our approach, we have analyzed 630 runs of ASPECTJ and compared the classification of JPURE to the output of PURITYKIT, a static purity analysis tool. In our experiments, 35% of the methods classified as impure by PURITYKIT never have side-effects, suggesting that there is a huge gap between static and dynamic purity. Our experiences with ADABU indicate that dynamic analyses do not need a static purity analysis but can also work well with dynamic purity information. The results for the mutability analysis indicate that dynamically parameters are almost always modified through the parameter reference rather than through aliased references.

In its current state, our tool induces a runtime overhead up to 420% for interactive programs. We have several ideas how to improve the performance of our tool:

Online Analysis Instead of writing trace data to the disk, we could analyze it on-the-fly and emit the results when the vm is shut down. Thus we avoid both writing the trace file and reading it again later.

Statistical Sampling The amount of trace data can be reduced by tracking only every n-th execution of a method. Since this technique affects the precision of the analysis, it has to be studied carefully before it is applied.

Compression Wang et al. [20] have successfully applied the SEQUITUR algorithm to compress trace data for dynamic slicing. This approach could also be used to compress the data traced by JPURE.

JPURE is publicly available so that other researchers can benefit from our work. For a detailed how-to, source-code and compiled versions please visit

`http://www.st.cs.uni-sb.de/models/jdynpur/`

Acknowledgments. Andrzej Wasylkowski and Andreas Zeller provided valuable comments on earlier revisions of this paper.

8. REFERENCES

[1] S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst. Combined static and dynamic mutability analysis. In *ASE '07: Proc. of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 104–113. ACM, 2007.

[2] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proc. of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41. ACM, 1979.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. *SIGSOFT Softw. Eng. Notes*, 27(4):123–133, 2002.

[4] N. Catano and M. Huisman. Chase: A static checker for jml's assignable clause. In *VMCAI 2003: Proc. of the 4th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 26–40. Springer-Verlag, 2003.

[5] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI '88: Proc. of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 57–66. ACM, 1988.

[6] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *WODA 2006: ICSE Workshop on Dynamic Analysis*, May 2006.

[7] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proc. of the 22nd IEEE/ACM Int. Conf. on Automated Software Engineering*, November 2007.

[8] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *Proc. of the 24th Int. Conf. on Software Engineering*, pages 313–324. ACM, 2002.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proc. of the ACM SIGPLAN 2002 Conf. on Programming language design and implementation*, pages 234–245. ACM, 2002.

[11] http://asm.objectweb.org/. ASM 2.2.1 GPL, 2008.

[12] http://columba.sourceforge.net/. Columba MPL, 2008.

[13] http://www ali.cs.umass.edu/DaCapo/benchmarks.html. Jolden, 2008.

[14] http://www.eclipse.org/. Eclipse GPL, 2008.

[15] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *POPL '02: Proc. of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–32. ACM, 2002.

[16] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in jit optimizations. In R. Bodik, editor, *Compiler Construction, 14th Int. Conf.*, volume 3443 of *LNCS*, pages 287–304, Edinburgh, April 2005. Springer.

[17] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. *SIGSOFT Softw. Eng. Notes*, 27(4):1–11, 2002.

[18] A. Rountev. Precise identification of side-effect-free methods in java. In *ICSM '04: Proc. of the 20th IEEE Int. Conf. on Software Maintenance*, pages 82–91. IEEE Computer Society, 2004.

[19] A. Sălcianu and M. C. Rinard. Purity and side-effect analysis for java programs. In *Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, pages 199–215, 2005.

[20] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing java programs. In *ICSE '04: Proc. of the 26th Int. Conf. on Software Engineering*, pages 512–521. IEEE Computer Society, 2004.

[21] M. Weiser. Program slicing. In *Proc. of the 5th Int. Conf. on Software engineering*, pages 439–449. IEEE Press, 1981.

[22] H. Xu, C. J. F. Pickett, and C. Verbrugge. Dynamic purity analysis for java programs. In *Proc. of the 7th Workshop on Program analysis for software tools and engineering*, pages 75–82. ACM, 2007.