Object Histories

Aakarsh Nair
University of Waterloo
aakarsh@gmail.com

Patrick Lam
University of Waterloo
p.lam@ece.uwaterloo.ca

ABSTRACT

Java programs manipulate objects by adding and removing these objects from collections and by putting and getting objects from other objects' fields. Convoluted object histories complicate program understanding by forcing software maintainers to track the provenance of objects through their past histories. This paper presents a novel approach which answers queries about the evolution of objects throughout their lifetime in a program. On-demand answers to object history queries aids the maintenance of large software systems by allowing developers to pinpoint relevant details quickly. We describe an event based, flow-insensitive, interprocedural program analysis technique for computing object histories and answering history queries. Our analysis technique uses pointer analysis to filter out irrelevant events in an object history and uses prior knowledge of the meanings of methods in the Java collection classes to improve the quality of the histories. We present the details of our technique and experimental results using open source Java benchmarks.

1. INTRODUCTION

Understanding large software systems is typically a difficult and time-consuming task. A key reason that large systems are difficult to understand is that systems contain masses of implementation details, making it hard to find a particular implementation detail needed to solve a problem at hand. In particular, programs often store objects to the heap and then later retrieve these objects. Java programs also often put objects into Java collection classes such as `ArrayList` and then later get objects back from these classes. To successfully understand the program, a developer must be able to trace the provenance of an object, not only through the code of a method at hand, but also back through its history on the heap.

In this paper, we propose *object histories*, which succinctly describe the evolution of objects in Java programs as they pass through the heap. Object histories summarize the events that an object undergoes throughout a program's execution. Events include instantiation, reads from other objects' fields or from collections, writes to object fields or collections, and being returned from a method call. Our event-centred approach eliminates the need to construct an large and unwieldy interprocedural program dependence graph. We instead use pointer analysis information to filter out irrelevant events from our object histories before presenting the results to the developer.

Our basic approach uses a field-based technique and stores a collection of program events. Events include heap reads, heap writes, uses of method return values, and object instantiations. Note that heap accesses include not only field accesses but also Java collection accesses; for instance, our system understands a call to `java.util.Collection.add` and treats it as a change to the appropriate object. Because collections are ubiquitous in Java programs, we believe that special treatment for accesses to collections greatly aids program understanding.

When an event references a heap location, our analysis abstracts that heap location by storing the fully-qualified field name with an abstraction of the base object.

Consider first a read from a field:

```
x = o.f.
```

Given a query for the history of `x`, our approach would return all events E which write to `o.f`, as well as all events which write to heap locations read by events $e \in E$.

We next extend the brief example with a field read: `o.f = x; y = p.f;` If we know that `o` and `p` may not alias, then we should not report the write `o.f = x` as part of the history of `y`. Of course, the same holds if `o` and `p` are collections. We therefore filter our query results using points-to information to remove irrelevant results.

Consider the following program:

```
1  class C {
2   Object f, g;
3
4   public static void main(String[] argv) {
5    o = new C(); m = new C(); y = new Object();
6    m.f = new Object();
7    o.f = y;
8    x = o.f;
9    p.g = x;
10   z = p.g;
11  }
12 }
```

We compute the history of `z` in `main()` as follows. We observe that `z` is assigned from a read of field `g` of an object of class `C`. Our field-based analysis tells us that the field write at statement 9 is related to the read of `C.g` at statement 10, so the value for `z` comes from the local variable `x`. The same local variable `x` is read from field `C.f`, which was written at line 7 from variable `y`. This branch of the history terminates at the instantiation of `y` at line 5. We filter out the other

write to C.f at line 6 because its base object, m, may not alias the base object o at line 8.

Object histories report all of the events that may happen to an object. Histories start from a given program point, and include all related program events (both in the past and the future, for objects that are stored on the heap at some point). In particular, object histories include field reads and writes as well as collection manipulations involving the object, and reach back to the original instantiation points of the object.

1.1 Contributions

This paper makes the following contributions:

- the notion of object histories, a collection of relevant events which affect an object throughout its lifetime in a software system;
- an algorithm for computing object histories and for filtering histories using pointer analysis information;
- an implementation of our algorithm; and
- experimental results from using object histories on a number of realistic benchmark programs.

2. MOTIVATING EXAMPLE

We next present a more in-depth example which explains our algorithms for computing sets of events as well as for querying these sets. Section 4 presents our experience with object histories on a number of real benchmark programs.

2.1 Basic Approach

Figure 1 presents an example program which we use to illustrate our approach. We start by presenting the basic approach, where we only use fully-qualified field names to match events; Section 2.2 explains how we leverage points-to analysis to improve the results of the basic analysis.

Figure 1 presents a short Java program which exhibits interesting heap and collection-based behaviour; in particular, it creates a pair of Objects, stores them in collections, and retrieves them again. To make our example even more interesting, we use helper methods to store and retrieve objects from the collections, thus necessitating interprocedural analysis techniques. Figure 2 presents the Jimple [**?**] intermediate representation code for main() from Figure 1; we have included it to show some of the temporary variables which occur in the subsequent figures.

Preprocessing Phase.

The first stage of our tool computes intraprocedural object histories for all non-library methods in the input program. In our example, we would say that local variable a in main() comes from the new Object() statement at line 22 in Figure 1, while the temporary local variable temp$5 comes from the call to n.addToL() at line 24. The intraprocedural histories underlie the events making up our object histories.

Second, we use the intraprocedural object histories to construct field and method summaries, which consist mainly of events. Figure 3 presents the events in our example. Field summaries group together all events which may change a particular field. For instance, Figure 3 contains a field WRITE event for line 7, as well as corresponding READ events of the same field at lines 10 and 13. Method summaries contain a set of events for each method, along with information about parameters and return values of the method. All of the events in Figure 3 from lines 19 through 29 belong to main().

Answering Queries.

The method and field summaries enable us to answer object history queries. Figure 4 presents (lightly reformatted) output from our object history browsing tool on a query about the x variable in the main() method. First, we use the intraprocedural object histories to report that x is the return value for call to the bar() method on line 29, emitting the first INVOKE event in the output. The summary for bar() informs us that it returns the return value from foo(). Next, we find that the method foo() returns the value returned by a call to a collection method, ArrayList.get().

We recognize the call to get() as a COLLECTION_READ call to a collection method and retrieve the collection by seeking information about the base object, temp$0 for method foo(). The intraprocedural object history informs us that temp$0 is a return value from the getList() method, which returns the list field from some object of class M (as seen in the READ event at line 13).

We can therefore match the COLLECTION_READ from the list field with the COLLECTION_WRITE event on line 10. We continue by investigating the source of the parameter a in addToL(). The intraprocedural object history informs us that a was formal parameter 0 to the addToL() method, so we need to consult all calls to addToL() and find out what the corresponding actual parameters were.

The call graph reports that there are two INVOKE events on addToL(), one at line 25 of main() and one at line 26 of main(). The provenance of actual parameter 0 in the call at line 25 was the NEW event on line 22, while the provenance of the parameter in the call at line 26 was the NEW event on line 23.

2.2 Filtering

During our preprocessing phase, we also record points-to sets when they are relevant to program events. These sets enable us to improve our responses to queries by discarding irrelevant information.

When we apply our filtering algorithm to the example from Figure 1, we obtain almost the same results as in Figure 4, except that we omit the call to n.addToL(). Note that our filtering is transitive: even though neither variable n nor variable m has any direct relationship with the queried variable x, our technique knows that only the calls where the receiver object of the addToL() call may alias the parameter to bar() are relevant.

We explain the operation of our filtered query. As before, we are retrieving the object history of variable x in the main() method. Once again, we traverse the return value information in our method summaries to reach the foo() method. Because our approach is context-insensitive, we have one points-to set for the possible arguments of foo(). This points-to set $pts(m)$ contains the object m instantiated at line 19. We then visit the event COLLECTION_READ. This time, we filter the matching COLLECTION_WRITE events and throw out the ones which have a base object that may not alias m, namely the call to n.addToL() at line 26 of

```
1  import java.util.ArrayList;
2
3  class M {
4    private ArrayList list;
5
6    public M() {
7      list = new ArrayList();
8    }
9    public boolean addToL(Object a) {
10     return list.add(a);
11   }
12   public ArrayList getList() {
13     return list;
14   }
15 }
16
17 public class Main {
18   public static void main(String[] args) {
19     M m = new M();
20     M n = new M();
21
22     Object a = new Object();
23     Object b = new Object();
24
25     m.addToL(a);
26     n.addToL(b);
27
28     Main main = new Main();
29     Object x = main.bar(m);
30     // x and a aliased after bar()
31
32     System.out.println(x);
33   }
34   public Object foo(M m) {
35     return m.getList().get(0);
36   }
37   public Object bar(M m) {
38     return foo(m);
39   }
40 }
```

Figure 1: Example Java Program.

```
1  public static void main(java.lang.String[])
2  {
3   java.lang.String[] args;
4   M m, temp$0, n, temp$1;
5   java.lang.Object a, temp$2, b, temp$3, x, temp$7;
6   boolean temp$4, temp$5;
7   Main main, temp$6;
8
9   args := @parameter0: java.lang.String[];
10  temp$0 = new M;
11  specialinvoke temp$0.<M: void <init>()>();
12  m = temp$0;
13  temp$1 = new M;
14  specialinvoke temp$1.<M: void <init>()>();
15  n = temp$1;
16  temp$2 = new java.lang.Object;
17  specialinvoke temp$2.<java.lang.Object: void <init>()>();
18  a = temp$2;
19  temp$3 = new java.lang.Object;
20  specialinvoke temp$3.<java.lang.Object: void <init>()>();
21  b = temp$3;
22  temp$4 = virtualinvoke
23             m.<M: boolean addToL(java.lang.Object)>(a);
24  temp$5 = virtualinvoke
25             n.<M: boolean addToL(java.lang.Object)>(b);
26  temp$6 = new Main;
27  specialinvoke temp$6.<Main: void <init>()>();
28  main = temp$6;
29  temp$7 = virtualinvoke
30             main.<Main: java.lang.Object bar(M)>(m);
31  x = temp$7;
32  return;
33 }
```

Figure 2: Jimple IR code for `Main.main()`.

7	NEW	(ArrayList)
7	WRITE	(M.list, temp$0)
10	READ	(temp$0, M.list)
10	COLLECTION_WRITE	(temp$0, a)
13	READ	(temp$0, M.list)
19	NEW	(M)
20	NEW	(M)
22	NEW	(Object)
23	NEW	(Object)
25	INVOKE	(m.addToL, a)
26	INVOKE	(n.addToL, b)
28	NEW	(Main)
29	INVOKE	(main.bar, m)
35	INVOKE	(m.getList)
35	COLLECTION_READ	(temp$0)
38	INVOKE	(foo, m)

Figure 3: Events in the Example Java Program.

```
Object history for method: <Main: void main(java.lang.String[])>
               and variable: x

... from INVOKE of method: <Main: java.lang.Object bar(M)> (ln 29)
Looking up Return Value for method: <Main: java.lang.Object bar(M)>.
... from INVOKE of method: <Main: java.lang.Object foo(M)> (ln 38)
Looking up Return Value for method: <Main: java.lang.Object foo(M)>
... from INVOKE of method: <java.util.ArrayList: java.lang.Object get(int)> (ln 35);
    * method is a collection method.
Looking up COLLECTION_WRITEs to collection temp$0 (2 steps):
(1/2) Resolving base temp$0 from instance method invocation
    <M: java.util.ArrayList getList()>
Looking up Return Value for method: <M: java.util.ArrayList getList()>
... READ from field: <M: java.util.ArrayList list>
(2/2) Found COLLECTION_WRITE call to collection (ln 10) with argument a.
    * a was formal parameter 0 to method: <M: boolean addToL(java.lang.Object)>
... from INVOKE of method: <M: boolean addToL(java.lang.Object)> (ln 25)
 Formal a was argument 0 in call.
 Variable is local: temp$2 = new java.lang.Object (ln 22)
... from INVOKE of method: <M: boolean addToL(java.lang.Object)> (ln 26)
 Formal a was argument 0 in call.
 Variable is local: temp$3 = new java.lang.Object (ln 23)
```

Figure 4: Unfiltered Object History for variable `x`.

`main()`.

3. TECHNICAL CORE

We compute object histories in a number of phases. First, we use a intraprocedural dataflow analysis to track objects within methods. Next, we compute the set of program events based on the intraprocedural history information. Our implementation serializes the object histories so that they may be viewed in an Integrated Development Environment. Developers may then query the events database by specifying a method and local variable; our analysis computes and returns the object history for the requested variable. Our approach also supports filtering: users may greatly increase the precision of the query results by selecting a particular object and rejecting history information that is provably irrelevant to (does not alias) the selected object.

3.1 Preprocessing

Our algorithm for computing object histories assumes that points-to analysis [**?**] results are available. We start with an intraprocedural history analysis, which gathers the history of each object-typed local variable in each method. We then organize the intraprocedural history information to support efficient queries.

Intraprocedural Histories.

Our analysis first computes intraprocedural object histories: that is, for each method m and each local variable v, we determine the possible sources of v. Our intraprocedural analysis is a variant of copy propagation which classifies sources for v into one of five types: 1) m's formal parameters (i.e. v is a formal parameter of m); 2) reads from fields (`v = o.f`); 3) return values from a call to method p (`v = p()` or `v = o.p()`); 4) new object instantiations (`v = new X()`); and 5) `catch` exception-handling blocks (`catch(v)`). Our analysis computes a set of sources for each variable v at each program point in m.

3.2 Computing Method and Field Summaries

We next use the intraprocedural history information to compute summaries for each method. A summary for method m consists of the following information: 1) a summary of heap changes effected by m; 2) information about the actual parameters of methods called by m; 3) information about the return values from m; 4) the intraprocedural history information for m; and 5) a list of callers of m.

Abstraction.

Our basic abstraction is field-based; that is, we combine information for all fields `f` of a class. (We found that our filtering technique, as described below, provides most of the advantages of an field-sensitive approach.) Our field-based approach, combined with filtering, is particularly effective for typical Java programs, which mostly access fields on the `this` object.

We also store local variables by name for use within methods.

Field Changes.

Our summaries include information about the field changes that each method effects. A field change includes information on the kind of change (e.g. `COLLECTION_READ`, `COLLECTION_WRITE`), a reference to the abstract object being mutated (the subject of the change), a reference to the abstract object being added or removed (the indirect object of the change), and any relevant additional method arguments.

Parameters and Return Values.

We record the identity of the parameters and return values with our method summaries by storing the relevant local variable names. The intraprocedural history information enables us to map the parameters and return values to their sources.

3.3 Answering queries

By combining intraprocedural object histories from several methods with pointer analysis information, our system provides useful information to developers about the provenance of their objects.

An object history query consists of a pair $\langle m, v \rangle$ where m is a method and v is a local variable belonging to m. Our system answers a query with a sequence of history events. We create four types of history events: object instantiations; objects passed back to callers as method return values; objects passed to methods as actual parameters; and field or collection accesses.

We combine local object histories to answer queries. Starting at method m, we report events on variable v. If v is a return value from method m', we retrieve the method summary for m' and recursively report events for the return value of m' using its history. Similarly, if v is parameter n of method m, we use the call graph to fetch a list C of callers to m and recursively report events for actual parameter n of each call $c \in C$ to m. Finally, if v was read from field `f` of class `K`, we report object histories for each write to field `f` of `K`. Similarly, if v was retrieved from a collection ℓ stored in field `g` of class `K`, we report histories for each addition to collection ℓ. (Our filtering algorithm helps eliminate irrelevant writes.)

3.4 Filtering

We have implemented a postprocessing phase which filters the analysis results based on pointer analysis information before returning the results to the user. To implement filtering, we augment our abstraction with all relevant points-to sets, and record them as we compute our intraprocedural histories and summaries. In response to a query, we filter INVOKE events which correspond to calls to instance invoke statements as well as READ fields corresponding to reads of instance fields. (Our histories also account for static methods and fields.)

Note that it is incorrect to ask the points-to analysis about receiver objects that may alias the query variable `v`. In particular, object histories record the evolution of a variable as it passes through the heap.

Consider an instance field READ `v = o.f` of field `f` belonging to class `C`. Our raw algorithm considers all WRITEs to field `C.f`. Filtering reduces this set of WRITEs by running a points-to analysis query on every write to `p.f = x` of field `C.f`, checking that `o` may alias `p`.

Object history queries encounter instance invokes `n.foo()` when traversing the intraprocedural history information, say for a method `m()`. But the intraprocedural history information also contains information about all of the callers to

m(), including points-to information about the parameters to m() at each of the sites calling it. Our filtering information rejects the call to n.foo() if it is inconsistent with the points-to information it has collected.

3.5 Remarks

We only analyze non-library classes of our application. In principle, library classes could affect object histories. However, we believe that object histories are still useful even if they omit the effects of library classes, for a number of reasons. First, we explicitly summarize the effects of collection classes, which are the likely to be the most-relevant library methods for understanding where objects come from. Second, the fact that we do not require the complete system dependence graph (as in program slicing), but instead combine events from different methods in a flow-insensitive fashion, is quite useful: arbitrary control-flow in library methods cannot affect our results. Our results will only omit writes to fields and collections by library classes. In general, library classes would not write to application fields, since they would not be aware of their existence. There is no conceptual barrier to analyzing library classes as well; we only chose to omit them to limit the size of our analysis results.

Because our approach is flow-insensitive, we sacrifice some information that would be available in program dependence graphs about statement ordering. In particular, if we request the history of a variable in method m, which is called by method n, then we cannot exclude the statements which happen in n after the call to m. That is, we simultaneously compute both the backward and forward slices as responses to our query.

4. EXPERIMENTAL RESULTS

We have implemented our analysis to compute object histories on top of the Soot program analysis framework [**?**], implemented a graphical tool to browse these object histories, and investigated the behaviour of our system on two benchmarks. Our benchmarks are the Gantt project and jGraphX. In this section, we describe our benchmarks and what we found out about them by using object histories.

4.1 GanttProject

Our first benchmark is version 2.0.10 of GanttProject, a GPLed tool for project scheduling and management implemented in Java. It consists of 64,000 lines of Java code in 482 classes.

We investigated the code for GanttProject and picked a typical method, getProject(), which belongs to the GanttXMLSaver class. It retrieves the value of the myProject field. Our tool gives the following output for this method:

```
Object history for method: <...GanttXMLSaver:
   net.sourceforge.ganttproject.IGanttProject getProject()>
              and variable: $r1

... READ from field: <...GanttXMLSaver:
      net.sourceforge.ganttproject.IGanttProject myProject>
Field was ASSIGNED PARAMETER 0 (ln 67)
r1 was parameter number: 0 to method: <...GanttXMLSaver: void <init>(...)>
... from INVOKE of method: <..GanttProject$ParserFactoryImpl: ...GPSaver newSaver()>
Variable is local: $r1 = new ...GanttXMLSaver (ln 2458)
```

We can see that the myProject field was written by the method newSaver of the anonymous ParserFactoryImpl inner class of GanttProject.

4.2 JGraph X

We have also run our object history tool on version 0.99.0.7 of the JGraph X graph drawing and visualization component. This benchmark consists of 42,500 lines of Java code in 82 classes.

The method setSelectionCell() from the mxGraph class contains a read from the selectionModel field. Our object history tool then browses the set of events and finds that this field was written to at line 472 of the mxGraph class, in the createSelectionModel() method. Furthermore, createSelectionModel() instantiates a new mxGraphSelectionModel at line 483. We can therefore see that our object histories successfully traverse the heap and find the instantiation site of an object created elsewhere in the program.

```
Object history for method: <....mxGraph: void setSelectionCell(java.lang.Object)>
              and variable: $r2

... READ from field: <....mxGraph: ....mxGraphSelectionModel selectionModel>
Field was ASSIGNED INVOKE <....mxGraph: ....mxGraphSelectionModel createSelectionModel()> (ln 472)
$r8 was assigned from INVOKE of method: <....mxGraph: ....mxGraphSelectionModel createSelectionMode
Looking up Return Value for method: <....mxGraph: ....mxGraphSelectionModel createSelectionModel()>
Variable is local: $r1 = new ....mxGraphSelectionModel (ln 483)
```

5. RELATED WORK

We discuss three main areas of related work: points-to analysis (and its applications to program understanding), program slicing, and program exploration approaches.

5.1 Points-to Analysis

Our approach relies on good points-to analysis information to filter query results. We use the default Spark implementation of points-to analysis due to Lhoták [**?**], which computes call graph and context-insensitive field-sensitive points-to information for Java programs.

Points-to analysis is useful for program understanding; for instance, Ghiya [**?**] describes how displaying the read and write sets associated with particular program points can help developers with program understanding. Object histories help make points-to information more helpful to developers by showing the evolution of abstract memory locations (points-to sets) over time.

5.2 Program Slicing

Object histories can answer queries about a program variable by identifying a set of program events potentially relevant to the history of that variable. Program slicing [**?**, **?**] attempts to identify a minimal subset of the program which affects a variable at a given program point.

A key difference between our approach and program slicing is that our approach is flow-insensitive at an interprocedural level. Typical slicing approaches compute a Program Dependence Graph, or, interprocedurally, a System Dependence Graph [**?**] and identify all statements which exert control or data dependencies on the variable being queried. Such dependencies are paths through the PDG or SDG. Our approach instead computes a set of events in the program, organized by method, and recursively queries this set of events to build object histories. We have chosen to define an event e to be relevant to a query on variable v if v is transitively data-dependent on e. Our flow-insensitive, event-based approach preserves much less state and hence makes queries significantly easier to answer: instead of needing to traverse the entire program, we can construct a list of relevant events by querying the suitably-indexed set of events. Furthermore, because object histories focus on summarizing the behaviour of reference-typed variables, and need not deal with scalar variables (which

program slicing typically devotes significant effort to understanding), we could use object-specific events such as instantiation and field reads in our summaries, and pointer analysis information was particularly meaningful for summarizing events. Our experience was that object histories, in conjunction with the flow-sensitive, context-sensitive pointer information, provided useful information about program behaviour.

Kaveri [**?**] is an Eclipse plugin providing a front-end to the Indus slicer for Java programs. Our filtering by pointer information, in particular, is similar to Indus's use of share entities [**?**]. Our results, however, indicate that detailed pointer analysis information eliminates much of the need for considering complex dependence graphs which summarize the whole program; instead, our object-insensitive, field-based approach combined with field-sensitive pointer analysis successfully identifies a useful set of events for understanding the behaviour of heap-manipulating Java programs.

Another important difference between our approach and typical program slicing is our special treatment of collection classes. Because Java programs use collections extensively, encoding the semantics of collection manipulations into our algorithms enables our histories to express what happens to objects over their lifetime more succinctly than it could otherwise.

5.3 Program Exploration Approaches

Demsky and Rinard present a technique for program exploration and understanding where object states depend on their dynamic membership in collections, or roles [**?**]. In their approach, a role represents a set of referencing relationships from and to a particular heap object; membership in collections is represented by inbound references to that object. They compute enhanced method interfaces by analyzing program execution traces. These enhanced method interfaces summarize the read and write effects of methods, similar to our intraprocedural object histories.

Another event-based program exploration and verification approach is the Program Query Language [**?**]. As with object histories, PQL also abstracts programs into a set of events and proposes a query language over this set of events and uses pointer analysis to improve the relevance of its query results. However, PQL was designed to support queries which find application errors and security flaws, and it therefore focusses on finding all points in a program which satisfy a query (i.e. execute a certain set of events), rather than displaying the events leading up to a query consisting of a specific program point.

6. CONCLUSION

In this paper, we have presented object histories, which enable developers to easily locate program statements relevant to a particular object-typed variable. Object histories operate on an event-based flow-insensitive abstraction of the program, enabling them to be easily stored and later retrieved; we anticipate adding support for browsing object histories to an Integrated Development Environment. Our analysis computes complete object histories for objects, which can summarize all events for an object, starting from the query point and reaching that object's instantiation points. Queries of an object history database are fairly straightforward and do not require nontrivial program analysis. Our experience with object histories on two nontrivial benchmarks suggest that they have significant potential to help developers navigate large programs.

7. REFERENCES

[1] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 313–324, New York, NY, USA, 2002. ACM.

[2] R. Ghiya. *Putting Pointer Analysis to Work*. PhD thesis, McGill University, May 1998.

[3] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 35–47, June 1988.

[4] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[5] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, 2005.

[6] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent java programs. In *Proceedings of Compiler Construction (CC'04)*, pages 39–56, 2004.

[7] V. P. Ranganath and J. Hatcliff. Slicing concurrent java programs using Indus and Kaveri. *Int. J. Softw. Tools Technol. Transf.*, 9(5):489–504, 2007.

[8] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[9] R. Vallée-Rai. Soot: A Java optimization framework. Master's thesis, McGill University, July 2000.

[10] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.