An Improved Slicer for Java

Christian Hammer
Universität Passau
Lehrstuhl Softwaresysteme
hammer@fmi.uni-passau.de

Gregor Snelting
Universität Passau
Lehrstuhl Softwaresysteme
snelting@fmi.uni-passau.de

ABSTRACT

We present an improved slicing algorithm for Java. The best algorithm known so far, first presented in [11], is not always precise if nested objects are used as actual parameters. The new algorithm presented in this paper always generates correct and precise slices, but is more expensive in general.

We describe the algorithms and their treatment of objects as parameters. In particular, we present a new, safe criterion for termination of unfolding nested parameter objects. We then compare the two algorithms by providing measurements for a benchmark of Java and JavaCard programs.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Algorithms

Keywords

Static Program Slicing, Java, Object Trees

1. INTRODUCTION

Slicing is a program analysis technique useful for many applications. Research in program slicing has produced systems such as CodeSurfer [19] or VALSOFT [7, 16], which can slice realistic programs written in the full C language with reasonable precision and performance. Naturally, one also wants to use slicers for C++ or Java. Such a slicer has to deal with objects, inheritance and dynamic dispatch.

One popular method to implement slicing is the construction of the *Program Dependence Graph* (PDG). The PDG represents the statements or predicates in a program as nodes, and possible influences as edges. Horwitz [5] extended this approach to interprocedural slicing; they introduced the *System Dependence Graph* (SDG) and developed a two-phase slicing algorithm for procedural programs.

PASTE'04, June 7-8, 2004, Washington, DC, USA.

Several extensions of the SDG to object-oriented features were proposed, and all of them were capable of handling dynamic dispatch and inheritance [9, 6, 24, 11, 23, 12, 22, 3]. Usually, dynamic dispatch is treated similar to function pointers in C [4]. Today, approximation of dynamic dispatch in slicers for object-oriented languages is reasonable precise and efficient, thanks to powerful call-graph and points-to analysis algorithms supporting SDG construction.

But there is another issue which has not been addressed properly. Very often, objects are passed as actual parameters to methods. But existing proposals for modeling objects as parameters are either insufficient to deal with recursive data structures, or they cannot distinguish field instances for individual objects (that is, dependencies concerning object fields are not object-sensitive). This makes slicing either incorrect (too small) or imprecise (too big).

In this paper, we present a new algorithm for dealing with objects as parameters, and compare it to the most sophisticated alternative published so far [11]. The latter introduced partial object-sensitivity for parameter objects, which increased precision as compared to these author's earlier algorithm [9]. Unfortunately, [11] is not explicit about the treatment of certain dependencies missed by the algorithm in [9], and experiences with an implementation have not been reported.

Our new algorithm is based on a safe criterion for termination of unfolding of parameter objects. It never misses any dependencies and is more precise than [11].

2. STRATEGIES FOR A JAVA SDG

In the following, we assume some general familiarity with slicing technology, as presented for example in [20].

Intraprocedural PDGs can easily be constructed for method bodies, using the well-known algorithms from the literature. Interprocedural slicing however is more tricky. While SDGs in general are well understood, dynamic dispatch and objects as method parameters make SDG construction more difficult. Treatment of dynamic dispatch is well known: possible targets of method calls are approximated statically (e.g. using points-to information), and for all possible target methods the standard interprocedural SDG construction is done.

Method parameters are another issue. SDGs support call-by-value-result parameters, and use one SDG node per in-

```
class A {
  int x, y;
  A(int i) { x = i; }
}

class B {
  A a;
  B() { a = new A(2); }
  int foo() { return bar(a); }
  static int bar(A a) { return a.x; }
  public static void main(String[] args) {
    B b = new B();
    int z = b.foo();
  }
}

class C {
  C f;
  public static void main(String[] args) {
    C c = new C();
    C.rec(c);
    C x = c.f;
    C y = x.f;
  }
  static void rec(C c) {
    c.f = new C();
    C x = c.f;
    x.f = c;
  }
}
```

Figure 1: An example program

resp. out-parameter. Java supports only call-by-value; in particular, for reference types the object reference is passed to the method. However field values stored in actual parameter objects may be changed during a method call. Such possible field changes have to be made visible in the SDG by adding modified fields to the formal-out parameters. This idea was already described by Larsen and Harrold [9].

Later, Liang and Harrold [11] proposed to improve precision by making the dependency analysis partially object-sensitive. They represent parameter objects as trees: the root of the tree is the actual parameter; the next level represents all fields which might be read or written in the method or in methods called in it. If such a field contains a reference to another object, its fields are added below the node representing that object and so on. Eventually, basic data types are obtained as leafs.

Liang proposed to restrict the tree depth for C++ objects and relies on points-to analysis for object references. This trick is called "k-limiting" [14]. It will miss dependencies between recursive data structures if no other (perhaps less precise) fallback mechanism for deeply nested objects is in place. Traditionally, k-limiting preserves soundness by providing an additional "summary node" which approximates all dependences for levels $> k$. However, [11] is not explicit about treatment of such dependencies; it only states that the simpler mechanism from [9] cannot even represent nested parameter objects.

This work presents a different approach: Instead of limiting the tree level, we unfold the tree completely. As this is not possible for recursive data structures, we present a *condition for safe termination of unfolding*. The condition is based on points-to information. This method keeps all trees finite but guarantees that no dependencies are lost. Points-to information is also used to constrain run-time targets of method calls. As a by-product, a call graph is extracted.[1]

But even the best points-to analysis will not resolve all object polymorphism, and the object trees must represent all possible run-time types of an object. In contrast to [11] we do not represent polymorphic objects as a set of trees, but as one "merged" tree. To disambiguate fields with the same name but defined in different classes we use the fully qualified field name. Thus merging does not reduce the precision of the final SDG; it just reduces its size.

Similarly, we use just one SDG node per method call even if dynamic dispatch is possible. For every method that – according to the call graph – might be called at runtime, we add a call edge to the entry of this method. Object parameters are represented as trees in the same way as for non-virtual methods. Thus we determine one actual-in tree node for every field that might be referenced in the (indirectly) called methods, and one actual-out tree node for every field that might be modified during method execution.

The method entry vertex analogously contains one formal-in tree node for every field which might be referenced, and one formal-out tree node for every field which might be modified in the method. For a virtual method call this means that there is not exactly one actual node for every formal node: Different (re)definitions of virtual methods may very well access a different set of fields of a parameter. Thus, every actual tree is a union of all corresponding formal trees of all possibly called methods in the approach presented here.

3. ALGORITHMIC DETAILS

Figure 1 presents a small Java program which will be used in the following technical descriptions.[2]

The computation of the trees representing parameter objects is similar to side-effect analysis [13]. Initially only the tree's roots are present at method entry and call nodes. Next, an intraprocedural analysis determines fields which are possibly defined or referenced in the method. Finally, trees are propagated interprocedurally through call sites and method bodies.

3.1 Intra-Procedural Step

Our analysis is based on an intermediate representation (IR) generated from bytecode. The IR contains the following quadruples (Quads) for field access:

$$dest = ref.field \quad \text{(GET)}, \qquad ref.field = source \quad \text{(SET)}^3$$

We iterate over these Quads until we reach a fixed point: For the object reference ref we determine the set al of aliased *input nodes* (formal-in or the actual-out SDG nodes modeling the returned value). For every node in al we assure that a child node f for the field $field$ is present. If a SET Quad

[1]The precision of the call graph depends on the precision of the points-to information, thus we cannot compare our call graph to those generated by e.g. XTA [21].

[2]In the following, the term *field* will mean non-static field if not stated otherwise. As usual, static fields are transformed into additional parameters.

[3]We handle the corresponding array Quads like an access to a field with name []. Array indices are ignored, which yields a conservative approximation.

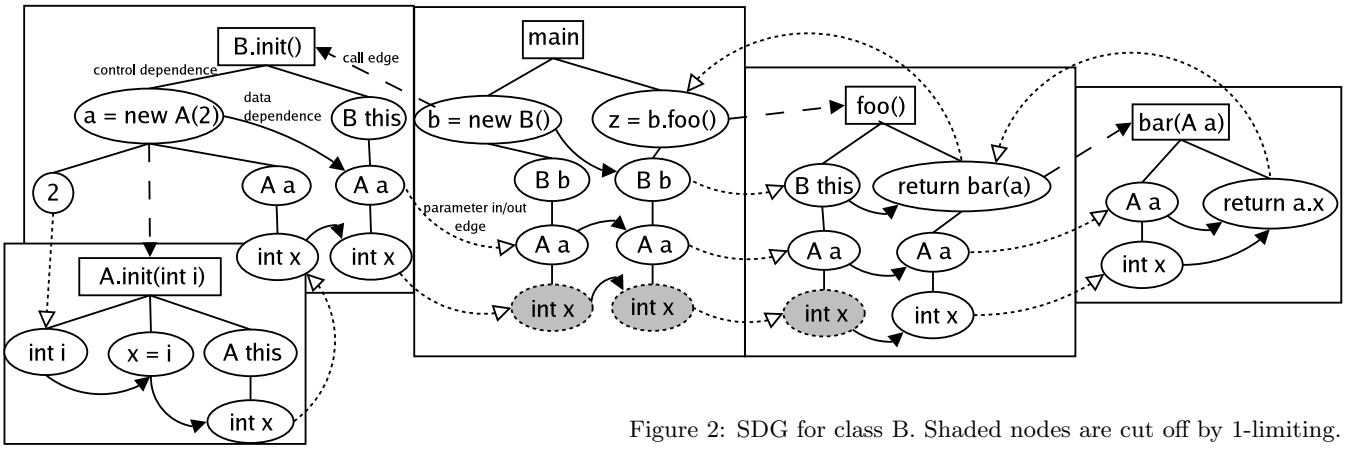

Figure 2: SDG for class B. Shaded nodes are cut off by 1-limiting.

is processed and the new node is part of a formal-in tree, we also have to add formal-out nodes to represent the induced side-effect. This is achieved by copying the whole path from r to f to the formal-out node corresponding to r (if it is not yet present we add one).

If a field which could be modified holds a reference to an object, all the fields of that object could be modified as well when the reference changes. Thus the latter fields must be added recursively to the object tree.

As a simple example, consider figure 2 (right). It shows the PDG for `B.bar` which returns the field `x` of the parameter object `a`. Thus, a field node has been added to the parameter node representing `a`. Furthermore, the field `x` is set in the constructor of class `A` (left in the figure). The (formal-out) nodes on the right have been added as a result of the corresponding SET instruction in the IR.[4]

3.2 The Unfolding Criterion

Liang and Harrold [11] already pointed out that in the presence of recursive data structures the object trees cannot be unfolded until all leaves are primitive types. As mentioned earlier, their solution, namely to limit the depth to a fixed level, is unsatisfactory. In our approach we unfold the tree until we reach a fixed point with respect to the aliasing situation of the containing object. Thus we obtain a safe criterion telling us whether further unfolding can be stopped without loosing dependencies. The criterion is based on points-to information and works as follows.

Criterion

Let $pt(x)$ be the points-to set for an object reference x. A node for field f need ***not*** *be added to a parent node p in the object tree, if the path from the root r to p contains another node $p' \neq p$ where $pt(p) = pt(p')$, and p' already has a child node for field f. If $pt(p) = pt(p')$, but p' does not yet have a child for f, f is added to p'.*

The correctness of the criterion becomes clear when considering how data dependences between field parameter nodes are computed. Every field parameter node represents a SET

[4]Note that there are no corresponding formal-in nodes on the left of this graph as the memory for the object is uninitialized before the call to the constructor.

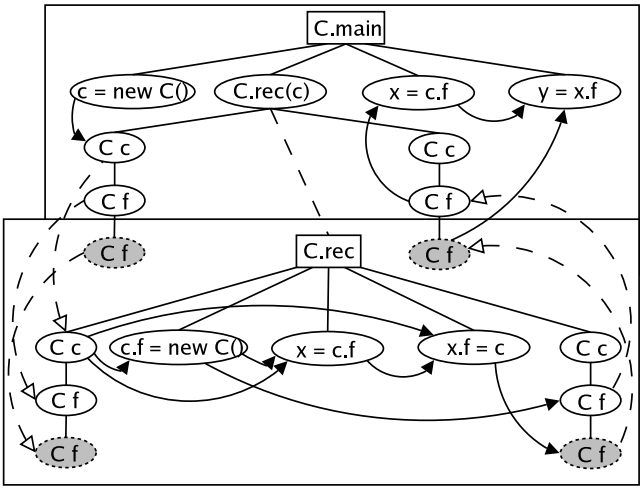

Figure 3: PDG for "main" including object trees.

of a field in a parameter object. A subsequent use of the field represents a corresponding GET. Data dependences between such fields are computed according to [5]: A PDG has a data dependency edge from node n_1 to node n_2 due to field dependencies iff all the following conditions hold:

- n_1 is a node that defines the field f of variable x
- n_2 is a vertex that uses the field f of variable y
- x and y are potential aliases
- control flow can reach n_2 after n_1 via a path in the CFG along which is no intervening definition of f and
- equality of the field f with respect to name and its defining class

This edge is then necessary for the PDG to be a conservative approximation. A definition of f is intervening iff there is a must-aliasing relation between x and the variable z which the field f in the definition belongs to (e.g. if x and z are the same variables or just renamed).

If a field node for f was added to p while $pt(p) = pt(p')$, then the data dependencies rooted in the f child of p' would

be identical to those rooted in the f child of p. Hence the f child of p can be omitted.

As an example, consider figure 3 (lower), which shows the PDG for `C.rec`. The points-to set of `c` contains only the `new` statement in `C.main`, while the points-to set of field `f` in `c` contains only the `new` instruction in `C.rec`. Both k-limiting and our algorithm add the first level field. The second level is added in our approach as the points-to sets of the root and the first level node are not equal. A third level is not added as the points-to sets of the `c` and `c.f.f` nodes are equal. The root already has a child node for `f`. Adding the field `f` a third time, would just result in the same dependences as `c` has because of the aliasing situation. In contrast, k-limiting would cut off the trees and miss the shaded nodes, hence dependences and slices would be incomplete.

3.3 Interprocedural Step

Until now we have only assured that a formal-in or actual-out parameter node exists in the SDG for all the fields referenced or written directly in the analyzed method. As there has to be an actual parameter node for every formal parameter node we have to copy 1. formal-in nodes to actual-in parameter nodes, 2. formal-out nodes to actual-out parameter nodes, and 3. the tree rooted in the actual-out parameter node representing the returned value to the formal-out parameter nodes modeling the return value of the called methods.

This can be seen in figure 2 where the formal-in parameter tree of `B.bar` has been copied to an actual-in parameter tree of the call to `bar`. The formal-out tree in `A.init` based in the `this` pointer has been copied to the actual-out tree of the constructor call in `B.init`. No object is returned in this example, so the node `z = b.foo()` needs not to be copied to the node `return bar(a)`.

After these trees have been copied, they represent either new references to fields (actual-in and formal-out) or they are new definitions (actual-out nodes representing side-effects). In both cases we have to propagate the new nodes intra-procedurally the way we did in section 3.1 but this time only the usages and definitions arising of parameter nodes have to be considered. After that, the propagated nodes have to be copied again. So we iterate over this inter-procedural step until we reach a fixed point.

In our example SDG, the actual-in node `int x` of `B.foo` is copied to the formal-in parameter node `A a` and then to the actual-in tree of the call to `b.foo` of `main`. Conversely, the actual-out node of `int x` in the constructor of `B` is copied to the formal-out tree in the constructor and then to an actual-out tree of the constructor call in `main`.

3.4 Data Dependence Computation

After the interprocedural step we have the complete signature, including modified (static) fields, of all methods called from `main`.[5] Thus all data dependences can now be computed easily.

[5]Static initializers have to be added as extra root methods if access to a field or a method of that class is encountered. Static variables defined or used in the root methods have to be connected assuming no particular execution order.

According to the definition of data dependence for fields, the definitions arising of SET-Quads and parameter nodes and uses arising of GET-Quads and parameter nodes are computed similarly to the procedural case. A fixed point iteration over the CFG reveals the reaching definitions. Aliasing information and the field name determine if there is a possible data dependence.

3.5 Implementation

We implemented our new algorithm, as well as the algorithm from [11], using the FLEX/Harpoon framework [1]. The FLEX points-to analysis was not used as it needs a pre-computed call graph; and as it is flow- and context-sensitive, it does not really scale. Thus we implemented a flow- and context-insensitive analysis following the proposals of Lhotak [10]. We do however use the FLEX SSA form and IR.

The call graph is based on points-to information and is used to approximate all methods which can be targets of dynamic dispatch. For every such method, summary edges are computed using the algorithm by Reps et al [15]. In order to improve precision, edges in object trees are not followed during the computation of summary edges. Instead, only the (already calculated) summary and data dependences leaving a formal-in node are included into the possible paths inducing a summary edge. Including tree edges would induce spurious summary edges between a parent node and nodes which are targets of its children's summary edges.

The object trees as described in section 3.2 and the object trees as described in [11] share the same implementation. The only difference is that for the Liang/Harrold version the trees are cut off at level 1 or level 2.

As of today, our slicer can handle full Java except threads. A well-known problem however are the libraries, which in Java lead to an excessive scaling problem: as of JDK 1.4.2, even the simplest "Hello world" program loads 248 library classes! Thus we currently leave out libraries, which can have the effect that dependencies through library code are missing (providing stubs for the whole API is very expensive and was not done so far). We did however provide stubs for the JavaCard language, which has a much smaller API.

The reader must be aware that our implementation does not have a fallback mechanism for cut-off parameter trees, making this particular implementation of k-limiting unsound (dependencies can be missing). Liang & Harrold most certainly did not have an unsound algorithm in mind, but unfortunately it is unclear what their implementation really does. We plan to integrate Liang's fallback mechanism as soon as we find out how it works.

4. EMPIRICAL EVALUATION

We measured average slice size for a series of benchmark programs (see table 1). Most programs are small student programs with an average size of 1kLoc; two are medium-sized JavaCard applets. The student programs use very few API calls, thus leaving out the libraries does not miss many dependences. The "Wallet" applet is from `http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html`, the "Purse" applet is from the "Pacap" case study [2]. Both applet SDGs contain all the JavaCard API PDGs.

Name	LOC	Nodes	Edges	Slices	Avg. size	Level 2	Level 1	Time	Summary
Enigma	922	2132	4740	724	661	638	631	5	1
Lindenmayer System	490	2601	195552	274	512	330	302	5	10
Union Find	1542	13169	990069	1440	9987	3205	2046	36	103
Plane-Sweep	1188	14129	386507	1006	3566	1739	1317	24	13
Network Flow	960	1759	3440	747	257	257	257	6	1
Library Admin	618	2281	4999	679	330	325	312	4	1
TSP	1383	6102	15430	1533	2187	2141	2033	15	2
SemiThue System	909	19976	595362	607	7855	780	529	24	33
JavaCard Wallet	252	21274	87726	3038	9201	8440	7179	16	19
JavaCard Purse	9835	184590	1484975	10620	109093	70093	55835	277	2338

Table 1: Data for benchmark programs

For every program, the LOC and SDG size (nodes and edges) is given, as well as the number of slices per program. Slices were selected by choosing a random SDG node as a starting point for a backwards slice. The average slice size (in nodes) is given for our algorithm as well as for the 1-limiting and 2-limiting. For the slice size, only true instruction nodes are counted in order to make the comparison independent from additional parameter nodes. The time for SDG construction is presented together with the time for summary edges. The latter is measured separately as it is based on an $O(n^3)$ algorithm.

There are no separate time measurements for k-limiting, as its implementation is based on the implementation of our algorithm. In fact we compare our algorithm to unsound k-limiting (where $k = 1$ or $k = 2$). This leads to missing dependencies between deeply nested fields, and our evaluation was designed to empirically check the effects.

The comparison reveals the following results. For half of the programs, the average slice size is roughly the same. A look at the source code reveals that these programs do not use many objects as actual parameters. For the other half of the programs, there are dramatic differences in average slice size: unsound k-limiting misses between 30% and 90% of the slice nodes determined by our algorithm – and the level 1 version behaves quite the same as the level 2 version. This indicates that many dependencies are due to fields in deeply nested parameter objects, making an unsound k-limiting quite problematic.

But could it be that unsound k-limiting looks so incorrect because our slicer is imprecise, hence the difference in slice size looks bigger than it really is? This argument is however not valid. Remember that in fact all three slicing algorithms as used in the comparison are based on the same implementation. The only difference is that either full trees are used, or trees are cut off at level 1 resp. level 2. All other imprecision due to points-to analysis, call graph construction, or other factors are the same in all algorithms. Thus the differences in slice size stem solely from the tree cut-off and its subsequent missing dependencies.

It would be very interesting to repeat the comparison using a version of the Liang/Harrold algorithm which uses an object-insensitive approximation for deeply nested objects, that is a sound k-limiting. Due to lack of time, we have not been able to perform this second evaluation.

5. RELATED WORK

One early approach to object-oriented slicing was the work by Larsen and Harrold [9]. This approach represents fields of object parameters as extra (scalar) parameters and thus merges all fields of different objects. This results in a more conservative approximation, as the approach is not object sensitive.

There were several proposals of Java implementations based on the Larsen/Harrold work: Kovács et al. [6] tried to implement a Java slicer based on that representation, which was slightly adapted for Java. Zhao [24] also bases his proposal on the Larsen/Harrold work. As already proposed by Malloy [12], he included "membership dependences" and "inheritance dependences". Eventually Liang [11] pointed out that Larsen's approach is insufficient: Fields passed to other method calls cannot be represented.

Walkingshaw [23] implemented a SDG generator for sequential Java using the SOOT framework. Exceptions are not yet represented. Like our approach, it is object sensitive for field dependencies, but no algorithm to compute the object trees was given. In addition to our SDG, his graph contains membership dependences and inheritance dependences as proposed in [12]. For non-executable slices, the latter are not necessary as they do not increase the precision of the slice. Thus in our SDG membership and inheritance dependences are omitted.

Tonella [22] already proposed to use the results of a flow-insensitive points-to analysis to resolve the runtime types of an object but did not differentiate the fields of distinct objects. Thus his approach is object-insensitive and lacks precision.

The Bandera [3] project uses a Java slicer to automatically reduce the size of the transition system for model checking of Java source code. The Bandera slicer is designed as a model checker front-end, not as a tool for program analysis.

CodeSurfer [19] contains an alpha version slicer for C++. A Java version is planned. As of today, nothing is known about the precision of Codesurfer for C++.

6. CONCLUSION AND FUTURE WORK

We presented a new slicing algorithm for Java, which includes all dependencies between fields of nested objects but is more precise than previous algorithms. Improving [11], we described how to compute the full tree structures for parameter objects, and presented a criterion for safe termination of the tree unfolding process.

Comparing our algorithm to earlier algorithms, it turns out that the earlier slicers are either less precise, or can miss a substantial amount of the dependencies induced by nested objects, if deeply nested field dependencies are not approximated properly. In this workshop contribution, we did however not yet include a comparison with an implementation of the algorithm from [11] which approximates deeply nested field dependencies.

It turns out that a good points-to analysis is a prerequisite for computing the object trees and applying our unfolding criterion. It can be employed to approximate the call graph of the analyzed program, and to identify possible aliasing situations. The better the results of the points-to analysis, the more precise slices will be. We are convinced that a context-sensitive points-to analysis must be used for maximal slicing precision, and are currently integrating such an algorithm.

Our work is not finished at this point. As mentioned above, we will repeat our experiments using an approximation for cut-off trees in the Liang/Harrold algorithm. Furthermore,

- we plan to use our slicer for analysis of information flow [17] in safety-critical systems;
- we plan to augment slices with path conditions as described in [16];
- we plan to integrate Krinke's slicing algorithm for multi-threaded programs [8];
- we plan to implement approximation algorithms for the full Java API.

Our slicer will be integrated into the VALSOFT project, which is a program analysis tool for information flow in safety-critical software [18]. We believe that such tools will become more and more important with the increasing acceptance of Java for safety-critical applications.

Acknowledgement. Jens Krinke and the anonymous reviewers contributed valuable comments.

7. REFERENCES

[1] C. S. Ananian. The static single information form. Master's thesis, MIT, September 1999. Tech. Report MIT-LCS-TR-801.

[2] P. Bieber, J. Cazin, A. E. Marouani, P. Girard, J.-L. Lanet, V. Wiels, and G.Zanon. The PACAP prototype: a tool for detecting Java Card illegal flow. In *Java Card Forum*, Cannes, France, Sept. 2000.

[3] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[4] R. Ghiya and L. Hendren. A shape analysis for heap-directed pointers in C. In *Proc. 23rd Principles of Programming Languages*, pages 1–15, 1996.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[6] G. Kovács, F. Magyar, and T. Gyimóthy. Static slicing of java programs. Technical Report TR-96-108, József Attila University, Hungary, 1996.

[7] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Univ. of Passau, Germany, 2003.

[8] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proc. FSE/ESEC*, pages 178–187, 2003.

[9] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, pages 495–505, 1996.

[10] O. Lhoták. Spark: A flexible points-to analysis framework for java. Master's thesis, McGill University, Montreal, Canada, February 2003.

[11] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[12] B. A. Malloy, J. D. McGregor, A. Krishnaswamy, and M. Medlkonda. An extensible program representation for object-oriented software. *ACM SIGPLAN Notices*, 29(12):38–47, 1994.

[13] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the international symposium on Software testing and analysis (ISSTA'02*, pages 1–11, Roma, Italy, 2002.

[14] S. Muchnick and N. Jones. *Program Flow Analysis*. Prentice Hall, 1981.

[15] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[16] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *Proceedings International ACM/IEEE Conference on Software Engineering (ICSE'02)*, pages 478–488, Orlando, FL, May 2002.

[17] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[18] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. Submitted for publication, 2003.

[19] T. Teitelbaum. Code surfer user guide and reference. Technical report, Gramma Tech Product Documentation, 2001. http://www.grammatech.com/csurf-doc/manual.html.

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

[21] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. OOPSLA*, pages 281–293, 2000.

[22] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. In *International Conference on Software Engineering*, pages 433–443, 1997.

[23] N. Walkinshaw, M. Roper, and M. Wood. The java system dependence graph. In *SCAM*, 2003.

[24] J. Zhao. Dependence analysis of java bytecode. In *Proceedings of the 24th IEEE Annual International Computer Software and Applications Conference*, pages 486–491, October 2000.