Speeding up context-, object- and field-sensitive SDG generation

Jürgen Graf
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
graf@kit.edu

***Abstract*—System dependence graphs (SDGs) are an established tool for precise interprocedural program analysis. We present new techniques for the efficient generation of SDGs for full Java, which are context-, field- and object-sensitive. We show that previous approaches to the generation of interprocedural dependencies for Java do not scale, as they interfere with the points-to analysis. Our new algorithm is based on the WALA framework and reduces time and memory consumption up to 90%, while maintaining precision.**

I. Introduction

System dependence graphs (SDG) have been developed over the last 20 years and became a standard device to model structures and dependencies in a program. They are the basis for multiple applications in program analysis, such as slicing [8], [10], debugging [13], testing [1] and model-checking [5]. SDGs and precise slicing are used in a flow-sensitive, object-sensitive and field-sensitive information flow analysis for full Java Bytecode [4].

The precision of SDGs is crucial to the overall precision of those applications. Many algorithms have been proposed to improve precision of the SDG for object oriented languages [3], [9], [10], [12]. Despite these efforts, the computation of precise dependence graphs is still challenging for programs exceeding a certain size. This work evaluates the scalability of state-of-the-art precise SDG creation for object oriented languages [3]. These precise SDGs introduce additional method parameters to achieve context and field-sensitivity. Liang and Harrold [10] as well as Hammer [3] suggested using trees to model objects passed as parameters, where a node represents an object and its children represent its fields. We show that these object trees cause severe performance problems, because their size is inversely proportional to the precision (and thus to the runtime) of the points-to analysis: An imprecise points-to analysis leads to huge object trees, declining the performance of SDG generation. In order to get the object tree sizes under control, one has to employ precise points-to analyses, which do not scale for larger programs.

The main contributions of this paper are:

1) Object graphs - an extension of object trees - that improve space and runtime of SDG generation for less precise points-to information by merging duplicate information in subtrees into a single representation.
2) An optimization of the interprocedural propagation for object graphs that replaces the mutually recursive algorithm of object trees with 3 non-alternating phases.
3) An evaluation that shows how the combination of parameter passing model and points-to analysis influences runtime and precision of SDG generation.

Our SDG creation tool is based on the program analysis framework WALA[1] and supports three parameter models - object trees, object graphs and the approach included in the framework. We evaluated dependence graph creation under various options in a benchmark consisting of 20 small (100LoC) to medium (60kLoC) sized programs. Four different points-to analyses provide various levels of precision for the initial parameter information. The results show that object graphs are the best choice for imprecise points-to analyses. In general points-to precision in most cases has a moderate influence on the overall precision[2] (around 4%, but up to 24%) of the dependence graph while it greatly influences the runtime.

Section II, III and IV contain the description of the new parameter passing model and the approach included in WALA. The evaluation of its influence on scalability and precision of the SDG generation are in Section V and VI. Section VII, VIII and IX conclude with a brief summary, an outlook on future improvements and references to related work.

II. System dependence graphs

A system dependence graph $G = (N, E)$ for program p is a directed graph, where the nodes in N represent p's statements and predicates, and the edges in E represent dependences between them [6]. The SDG is partitioned into *procedure dependence graphs* (PDG) that model single procedures. In a PDG, a node n is *control dependent* on node m, if m's evaluation controls the execution of n (e.g. m guards a conditional structure). n is *data dependent* on m, if n may use a value computed at m. The PDGs are connected at *call sites*, consisting of a call node c that is connected with the entry node e of the called procedure through a *call edge* $c \rightarrow_c e$. Parameter passing and result returning, as well as side effects of the called procedure,

[1]IBM. T.J. Watson Libraries for Analysis (http://wala.sf.net/).

[2]The precision is measured through the average size of a slice in the SDG.

are modeled via synthetic *parameter nodes* and *edges*. For every passed parameter there exists an *actual-in node* a_i and a *formal-in node* f_i that are connected via a *parameter-in edge* $a_i \rightarrow_{pi} f_i$. For every modified parameter and returned value there exists an *actual-out node* a_o and a *formal-out node* f_o that are connected via a *parameter-out edge* $f_o \rightarrow_{po} a_o$. Formal-in and formal-out nodes are control dependent on entry node e, actual-in and actual-out nodes are control dependent on call node c. The parameter passing model guarantees that all interprocedural effects of a procedure are propagated via call sites. This well-formedness enables computation of summary information of interprocedural effects: So-called *summary edges* between actual-in and actual-out nodes of one call site represent transitive flow from a parameter to a return value in the called procedure.

SDGs permit to analyze program properties via graph traversal. Moreover, they are purpose-built for *context-sensitive* analyses, which distinguish different invocations of the same procedure.

A. SDGs for object-oriented languages

Our work focuses on the generation of SDGs for object-oriented languages. Challenges arising from object-oriented languages are *object- and field-sensitivity*, *exceptions*, *dynamic dispatch* and *objects as parameters*. These features lead to statically undecidable problems that are commonly approximated with the help of a *points-to* analysis. A points-to analysis statically analyzes a program's heap manipulations and determines to which objects a given reference[3] may point to at runtime. Based on this analysis one can derive related properties, like *may-aliasing*, i.e. whether two references may refer to the same object. One usage of points-to information is to resolve dynamic dispatch. Once it is known to which objects a reference may point to at runtime, one can determine the possible target methods of a dynamic dispatch. Points-to information is also used to achieve object- and field-sensitivity: It is possible to model two objects of the same type separately in the SDG if they are not may-aliasing.

Another application of points-to information is the precise computation of method side-effects. It is used to determine which object fields may be read or modified during method execution and to create the synthetic parameter nodes for these fields in the SDG. The computation of these synthetic nodes has a big influence on the scalability of the SDG generation, which is investigated in the subsequent section.

III. Improving Scalability

SDGs are built through an intraprocedural phase that covers local control and data dependencies for each method in isolation and an interprocedural phase that combines the intraprocedural results and models their effects on the global state of the program. The interprocedural phase consists roughly of three steps: (1) computation of additional parameter nodes arising from method side-effects, (2) computation of data dependencies for these new parameter nodes, and (3) summary edge computation [6], [11]. Steps (2) and (3) depend on the results of step (1), because they have to consider the additional nodes during their computation.

Our evaluation revealed that the intraprocedural phase has no scalability issues, whereas the interprocedural phase is the main reason for long runtime and huge memory consumption. The first step of the interprocedural phase seems to be crucial, because its results are used in the subsequent two steps. Employing the object tree parameter passing model, the SDG computation with a faster and less precise point-to analysis often leads to greater memory consumption and a longer overall runtime. We determined the way how additional parameters are computed to be responsible. In general a less precise points-to information resulted in a larger number of additionally created parameters and this slowed down the subsequent data dependency and summary edge computations. A closer look at the computation of those additional parameters will explain the observed behavior.

A. Parameters as object trees

Object trees are used as an object- and field-sensitive representation for all object fields a method may read or modify. A method holds an object tree for each parameter: The parameter itself corresponds to the root node of its object tree, while the child nodes match the accessed fields of the parameter object. The computation is initialized with a single root node for each parameter and subsequently adds new child nodes through a fixed point computation that consists of two mutually iterating steps: Step 1 examines each instruction in the current method and adds new child nodes to the current trees that match the accessed fields. Step 2 propagates the effects of method calls interprocedural. As step 2 may add new nodes to the object trees that leads to new nodes in step 1, both steps are repeated until a fixed point is reached.

1) Unfolding recursive data structures: A problem of this approach is that recursive data structures can lead to trees of infinite depth. Figure 1 shows an example, where a node contains a reference to a successor node. The size of the resulting list of nodes is in general statically not determinable, but an object tree has to be limited somehow. Liang and Harrold [10] proposed to k-limit the tree depth with an arbitrary number k. This approach prevents an infinite expansion, but may sacrifice soundness. Hammer [3] introduced a *unfolding criterion* that limits the size of object trees and maintains soundness without the loss of precision. It uses points-to information to decide how far an object tree is allowed to expand. The unfolding criterion works as follows: One only adds a child c to the tree, if no other node

[3]We use the term 'reference' for both references and pointers.

```
1  class Node {
2    Node next;
3
4    static void append(Node node, int len) {
5      node.next = new Node();
6      node = node.next;
7      for (int i = 1; i < len; i++) {
8        node.next = new Node();
9        node = node.next;
10     }
11   }
12 }
```

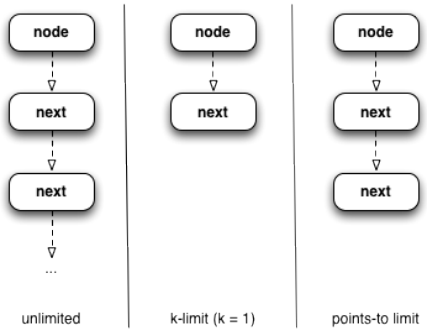

Figure 1. Example for object trees with potential infinite depth.

n on the path from the root node to the would-be parent node of c has the same points-to set as c.

In our example in Figure 1 any fairly precise points-to analysis is able to distinguish between the node object created in line 5 and the one created in line 8. Depending on the value of parameter `len`, line 8 is executed multiple times and thus may create more than one new object. The points-to analysis is not able to distinguish them, because the number of loop iterations is in general not decidable for a static analysis. Only the nodes created by the first statement can be distinguished from nodes created by the second statement. So the unfolding criterion limits the depth of the object tree for parameter `node` to two children. The right side of Figure 1 displays how the object tree for `node` looks like using the k-limiting from Liang and Harrold and the points-to limited unfolding criterion from Hammer.

2) Problem with imprecise points-to information: Object trees grow whenever two fields are may-aliasing. Thus a less precise points-to analysis leads to bigger object trees. This prevents usage of object trees in SDGs for large programs, as they tend to contain more may-aliasing fields: First, even the most precise points-to analysis has to approximate certain undecidable situations leading to may-aliases. Second, precise points-to analyses do not scale for arbitrary programs. Despite many improvements on the scalability of precise points-to analyses over the last years, they still remain expensive. So certain situations demand a less precise points-to analysis.

The upcoming example explains the computation of object trees and shows why imprecise points-to information leads to bigger object trees. We compare two SDGs for the program in Figure 2, one with precise points-to information and the other with less precise information. In this example, method `indirectMod` takes two arguments `x` and `y`. The statement in line 9 indirectly modifies a field value of parameter `x` through a call to method `modify` and the statement in line 10 returns a field value of parameter `y`. Both statements seem to be independent from each other, but that is not the case if `x` and `y` or their fields refer to the same object. We assume that a precise points-to analysis is able to distinguish the two objects passed to `indirectMod` in line 14 and their `a`-fields. The less precise points-to analysis distinguishes the two parameters, but not their `a`-fields (i.e. they are deemed to be may-aliasing).

The SDG on the left side in Figure 3 was built with the precise points-to information[4]. Without may-aliasing of the `a`-fields, the call to method `modify` affects only field `x.a.i`. Inside `modify`, line 6 needs to read field `b.a` at first to get the location of the `a` base object, before the field `i` can be written. So there are two input parameters: Object tree root node 20 for parameter `b` and child node 21 for the field `a` of parameter `b`. The output parameter nodes do not only consist of nodes for each modified field, they also contain nodes for all fields that have been read to access the base object of the modified field. These nodes represent the *access path*. The access path of node 24 thus is 22→23→24. The actual-in and -out nodes 15-19 at the call site of `modify` match the formal parameter nodes of the called method. Since `x` is passed to `modify` as parameter `b`, all fields of `b` are connected with the matching fields of `x` at the call site. The formal-in and -out nodes 6, 7 and 12-14 of `indirectMod` capture these side-effects on `x` and propagate them further to the `main` method. Using the precise points-to information it can be detected that the return statement in node 3 is independent from parameter `x` and only reads field `a.i` of the second parameter `y`. The formal-in nodes 8-10 correspond to this field accesses and formal-out node 11 represents the return value.

Less precise points-to analysis cannot distinguish between the `a`-fields and thus cannot determine exactly which field is accessed by the statements in line 6 and 10. So the SDG on the right side in Figure 3 takes both possibilities into account to yield a conservative approximation of the program behavior. The nodes that may refer to the same object are shaded gray and the new parameter nodes that had to be added as well as the additional dependencies are drawn with a thicker line. This SDG has four additional parameter nodes: One new formal-in node 25 and three

[4]For space reasons, the depicted SDGs exclude `main`.

```
1  class B {
2    A a = new A();
3  }
4
5  static void modify(B b) {
6    b.a.i = 42;
7  }
```

```
8  static int indirectMod(B x, B y) {
9    modify(x);
10   return y.a.i;
11 }
12
13 public static void main(String argv[]) {
14   indirectMod(new B(), new B());
15 }
```

Figure 2. Example for interprocedural parameter computation.

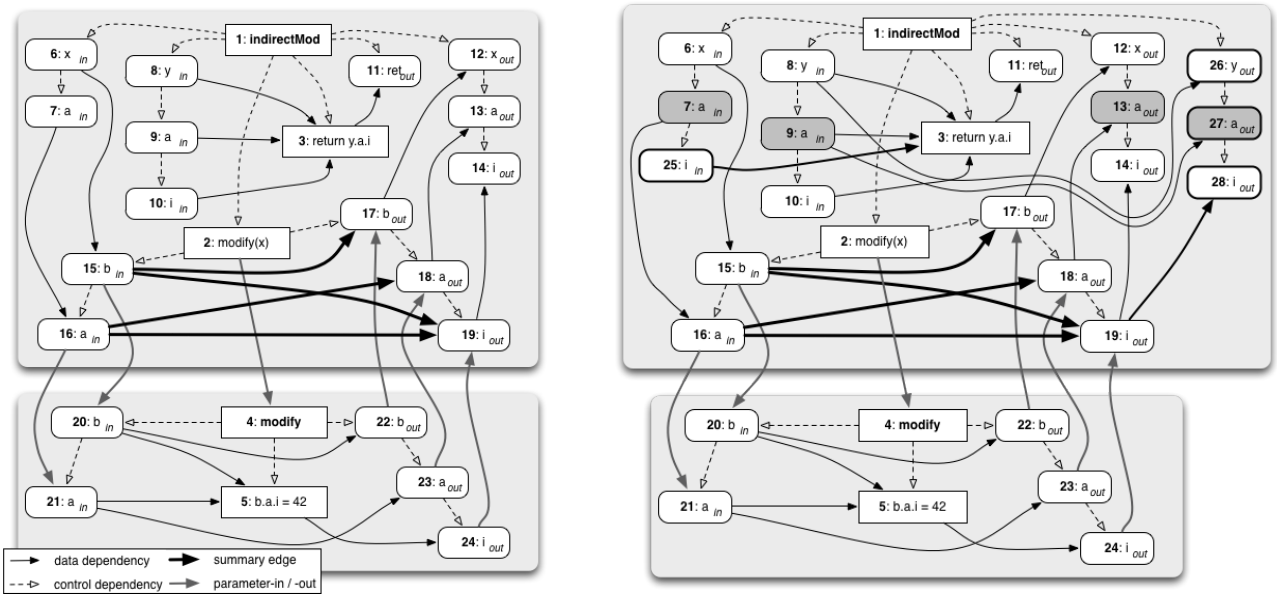

Figure 3. SDGs with object tree parameters for the program in Figure 2. Control dependencies are drawn with a dashed line and data dependencies use a solid line. The SDG on the left side was built with a more precise points-to information than the one on the right.

formal-out nodes 26-28. Node 25 captures the possibility that the return statement in node 3 may read the value of `x.a.i`, when `y.a` and `x.a` refer to the same object. The call to `modify` modifies the value of `x.a.i`, so the value of `y.a.i` may have been changed too. The formal-out nodes 26-28 model this new possibility. In this tiny example, the imprecise points-to information has led to 4 additional parameter nodes; our evaluation reveals that this effect is much worse for larger programs.

B. Parameters as object graphs

Object graphs are an extension of object trees. They also use the method parameters as root nodes and add child nodes for accessed fields. Their main difference is that parameter node subtrees are shared between nodes, when they cannot be distinguished by the points-to analysis.

An object graph node n is identified through a set of parent nodes, an object field and the points-to set of the field: $n := (parents, field, pts)$. An object tree node n' instead has only a single parent node together with an object field and its points-to set: $n' = (parent, field, pts)$. Multiple object tree nodes are mapped to a single object graph node as follows.

$$
\begin{aligned}
&n_1, n_2 \in Tree : \\
&n_1.parent.pts \text{ may-alias } n_2.parent.pts \wedge n_1.field = n_2.field \\
&\quad \Longrightarrow \\
&\exists_{=1} n \in Graph : n.field = n_1.field \ \wedge \ \exists p_1, p_2 \in n.parents : \\
&\quad p_1.field = n_1.parent.field \ \wedge \ p_1.pts = n_1.parent.pts \wedge \\
&\quad p_2.field = n_2.parent.field \ \wedge \ p_2.pts = n_2.parent.pts
\end{aligned}
$$

In the example from the right side of Figure 3 the nodes 7 and 9 as well as 13 and 27 are may-aliasing. Their child nodes 10, 25 and 14, 28 are merged into a single node for each pair. Figure 4 shows the corresponding object graph for the example with less precise points-to information[5].

Tree sharing does not affect the precision of the SDG, because nodes are only shared when they cannot be distinguished through the points-to information and thus have the same incoming and outgoing dependencies. It removes the scalability conflict between the points-to analysis and

[5]The object graph SDG with precise points-to information is not different from the SDG with the object tree parameter on the left side of Figure 3.

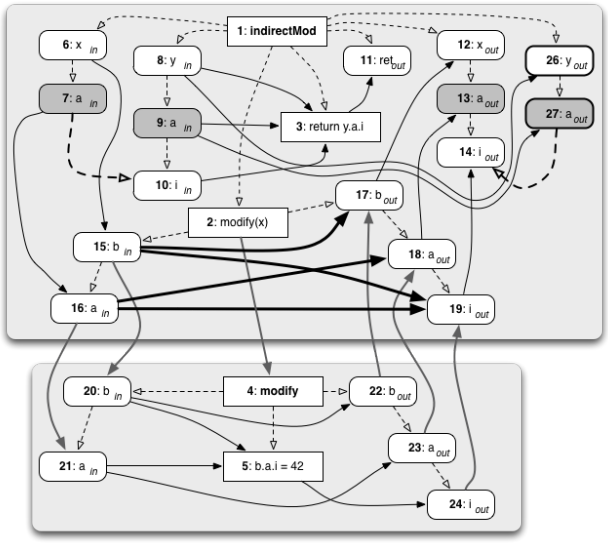

Figure 4. SDG with object graph parameters for the program in Figure 2.

the parameter computation of the object tree model. A less precise points-to analysis leads to more shared subtrees and a lesser number of nodes. The structural difference to object tree parameters is smaller for precise points-to analyses, where lesser may-aliases occur. The number of new parameter nodes from the object graph approach is nevertheless always smaller or equal to the number of nodes from the object tree approach.

C. Object graph optimizations

The shared subtrees help to decrease the number of nodes in case of imprecise points-to information, but the computation of object graphs suffers from another problem: It is slowed down by a fixed point solution that mutually iterates between intraprocedural and interprocedural phases. Thus our version of the object graph algorithm includes further improvements aside from tree sharing. The mutual dependencies between these phases can be removed with two modifications to the object graph model: (1) Each node is allowed to have multiple child nodes that refer to the same field. This modification breaks the property that an object graph always has a less or equal number of nodes then the corresponding object tree. However in practice this is only the case for very precise points-to information. (2) New nodes are created without checking for a matching parent node. So the graph may contain unreachable nodes with no existing access path.

These modifications enable us to split the computation of object graphs into 3 non-alternating phases: (1) Build intraprocedural nodes for each field access operation in the current method, (2) propagate the accessed fields interprocedural from callee to caller, (3) refine computed parameters by removing unreachable nodes with no access path.

1) Intraprocedural field accesses: The unoptimized object graph version inspects each field access and adds a single child node to all nodes that qualify as parent. Whenever a child node for the accessed field already exists, the existing node merges the points-to information of the new one. So existing nodes may change during the computation. We circumvent this behavior and create immutable parameter node candidates. A candidate contains the points-to set of the base object, the name and type of the field as well as the points-to set of the field itself: $(pts_{base}, field, pts_{field})$.

The program in Figure 2 contains 4 field accesses. Two in line 10 and two in line 6. The statement in line 6 of method `modify` reads the object where field `b.a` is pointing to and then modifies its field `i`. This leads to two parameter candidates[6] - one for the read access c_1 and one for the modification c_2: $c_1 = (pts(\texttt{b}), \texttt{B.a}, pts(\texttt{b.a}))_{ref}$ and $c_2 = (pts(\texttt{b.a}), \texttt{A.i}, pts(\texttt{b.a.i}))_{mod}$. Line 10 of method `indirectMod` contains two subsequent read accesses. It reads the object in field `y.a` and then it reads the field `i` of this object. They result in two parameter candidates: $c_3 = (pts(\texttt{y}), \texttt{B.a}, pts(\texttt{y.a}))_{ref}$ and $c_4 = (pts(\texttt{y.a}), \texttt{A.i}, pts(\texttt{y.a.i}))_{mod}$.

The points-to sets help to build the object graph structure from these candidates. For example c_2 may be accessed through c_1 and thus c_2 is a child node of c_1, because the points-to set of the field of c_1 is the same as the points-to set of the base object of c_2: $pts(\texttt{b.a})$. The unfolding criterion is met automatically, as the candidates are identified through points-to information and thus two candidates with the same points-to sets referring to the same field cannot exist.

2) Interprocedural propagation: The interprocedural step computes the effects of method invocations: The candidates of each method are propagated along each call site from callee to caller. This step does not introduce new or change exiting candidates. So their number is constant and a fast data flow analysis with bit vectors suffices for the propagation[7]. In the program from Fig. 2, all candidates of method `modify` are propagated to method `indirectMod`, as `indirectMod` calls `modify`. The previous intraprocedural step created candidates c_1 and c_2 for method `modify` and c_3 and c_4 for `indirectMod`. After the propagation method `indirectMod` contains all four candidates: c_1, c_2, c_3 and c_4.

At the end of the propagation each method holds a set of parameter node candidates. A formal-in parameter is created for each candidate of a read access and a formal-out parameter for each candidate of a write access. Additional formal-out nodes arise from candidates of read accesses that may be used to reach a modified field. The candidates c_1, c_3 and c_4 from our example lead to the formal-in nodes 7, 8 and 9 of Fig. 4. Formal-out node 14 stems from the

[6]We refer to the points-to set of a variable or field with $pts(name)$, where $name$ is the name of the field or variable.

[7]See our technical report 2009-14 for further details.

candidate c_2 and nodes 13 and 27 have been added for c_1 and c_3 as they can be used to reach the modified field[8].

Finally we connect all parameter nodes and build the object graph structure. Node a is connected to node b if the type of a contains a field with the name and type of b and if the field points-to set of a is not disjunct from the points-to set of the base object of b. The resulting object graph structure can then be exploited for further refinements.

3) Refinement with access paths: The object graphs are already usable after the interprocedural propagation, but their precision can be improved. They may contain nodes referring to field accesses that will never be visible outside the scope of the method they belong to. These nodes can be found by searching through their predecessors in order to find a path to a node of the root set. The value of a field node can only be reached from outside the method, if an access path from a root node to the field node exists. Otherwise the effects of the field access cannot escape the current method and can be removed from the object graph. In the example from Fig. 4 every node has at least one access path to the method parameter `x`, `y` or `b`. So no nodes are removed. This is very rarely the case in practice: Most methods contain unreachable nodes that cannot escape the scope of the method.

Our evaluation shows that those non-escaping nodes often introduce additional dependencies and therefore have a negative effect on the overall precision of the SDG. They should always be removed as far as the precision of the points-to analysis allows. Nevertheless, this step is optional and the object graph still remains a correct and conservative approximation without the refinement phase.

IV. Other Approaches to Interprocedural Data Dependencies

Another way to compute additional parameter nodes for data dependencies through fields is to create a parameter for each distinguishable memory location. This approach is used by the SDG generator that comes with the WALA framework. Its main advantages are that its implementation is straight forward and the computation scales quite well in theory. But it does have some drawbacks when it comes to precision and the number of additional parameters. While the additional parameters can be computed very fast, their number can become large and present a struggle for the successive phases of the interprocedural SDG computation, especially the summary edge computation. This approach uses an interprocedural mod-ref analysis to to detect a set of abstract location equivalence classes that may have been read or modified. The elements of this set then are used as additional parameters. The number of distinguishable location equivalence classes is directly related to the precision of the underlying points-to analysis and thus a more precise points-to analysis leads to a larger number of additional parameter nodes. There is also no further refinement of these additional parameters that removes unreachable nodes, like access paths in the object tree and graph approach.

V. Analysis Framework

We implemented our precise SDG computation on top of the publicly available WALA program analysis framework. WALA uses an intermediate representation in static single assignment (SSA) form that is very close to actual Java Bytecode. It includes its own SDG implementation and comes with multiple points-to analyses. It supports explicit and implicit exception flow, which helps us to maintain a sound approximation of the program. We incorporated WALAs call graph and points-to analysis as well as its intermediate representation into our implementation. We choose not to extend its SDG and instead built our own implementation with a focus on an exchangeable parameter model[9]. Nevertheless we integrated WALAs parameter passing scheme into our SDG generator and evaluated the differences to the our solutions.

Another popular program analysis framework for Java is SOOT [14]. SOOT uses also an intermediate representation in SSA-form and comes with control flow, call graph and points-to analyses. But it does not have its own SDG implementation and -to our best knowledge- does not support implicit exception flow. Thus we chose to use WALA.

Our tool supports a total of four different points-to analyses. From least to most precise they can be characterized as follows: A (1) *context insensitive type based* analysis that disambiguates objects according to declared types, a (2) *context insensitive instance based* analysis that disambiguates objects according to instantiation sites, a (3) *context insensitive instance based* analysis with *context-sensitivity for Container* classes that allows unlimited context-sensitivity for methods of objects implementing the Java Collection interface and finally an (4) *object-sensitive instance based* analysis that includes object-sensitivity for all instantiation sites.

VI. Experiments

We evaluated the performance and precision of SDG creation[10] on 20 Java programs with up to 4 different points-to analyses (Section V) and three different parameter passing models: The WALA approach using a single parameter per abstract heap location (Section IV), the object tree

[8] c_3 only results in a new formal-out node for the less precise points-to information. The less precise information cannot distinguish pts(`x.a`) from pts(`y.a`) and as `x` is passed to `modify` as parameter `b` we also know that pts(`x.a`) = pts(`b.a`). So `b.a` may refer to the same location as field `y.a` and thus c_2 may be reached through c_3.

[9] A rewrite of the SDG generator also enables us to include other features, like thread interference detection, into our existing tools more easily.

[10] We executed all tests on a computer with an AMD Opteron(tm) 8220 processors and 32GB of RAM running Ubuntu Linux 7.10 with Java 6 64-bit.

JC	CorporateCard	Purse	Wallet	Safe
LoC	485	5.066	124	577
WALA				
(1)	7.757 (63.36%)	44.022 (83.42%)	9.795 (66.07%)	7.488 (63.36%)
(2)	9.080 (61.89%)	163.700 (83.45%)	10.292 (64.82%)	9.247 (61.88%)
(4)	19.541 (59.68%)	193.427 (86.30%)	26.368 (63.29%)	20.558 (59.68%)
Tree				
(1)	8.463 (68.01%)	154.467 (77.74%)	9.870 (71.30%)	8.683 (66.92%)
(2)	8.311 (66.10%)	100.711 (76.63%)	10.595 (68.94%)	9.136 (66.10%)
(4)	20.115 (66.23%)	108.976 (76.55%)	22.178 (69.24%)	17.167 (66.18%)
Graph				
(1)	7.105 (51.78%)	32.247 (72.22%)	9.332 (55.63%)	6.046 (51.78%)
(2)	5.913 (50.63%)	45.835 (71.89%)	8.579 (54.81%)	6.078 (50.63%)
(4)	18.454 (48.58%)	197.225 (71.62%)	26.222 (52.10%)	21.702 (48.58%)

Table I
RUNTIME AND PRECISION STATISTICS FOR THE JAVACARD EXAMPLES.

J2ME	Barcode	J2MESafe	KeePassJ2ME
LoC	3.462	2.237	4.984
Graph			
(1)	106.380 (63.13%)	160.751 (57.92%)	685.791 (66.13%)
(2)	127.610 (51.79%)	200.733 (56.66%)	953.890 (64.24%)

Table II
RUNTIME AND PRECISION STATISTICS FOR THE J2ME EXAMPLES.

(Section III-A) and the optimized version of the object graph (Section III-B) model. All programs were analyzed including the library methods they used. Native methods were conservatively approximated through hand written method stubs. The parts included from the runtime library tend to be big and have an enormous impact on the runtime of the analysis. But when left out, the result of the analysis is no longer a conservative approximation.

The example programs vary in program size as well as in the size of the runtime library they use: 4 programs (Corporate Card, Purse, Wallet and Safe) are written for the small JavaCard[11] library with about 500LoC. Another 3 programs (Barcode, J2MESafe and KeePassJ2ME) use the J2ME library[12] with about 30kLoC. The remaining 13 programs (Battleship, HSQLDB and the 11 examples of the JavaGrande suite[13]) use the Java 1.4 library with about 100kLoC.

We canceled the computation when the memory usage exceeded 30GB or the runtime was longer than 2 days. The JavaCard programs as well as the small Battleship program could be analyzed with all available options. They are used to compare the effects of the different parameter models and points-to analyses. All J2ME programs and the big HSQLDB program could only be analyzed with the object graph model and the two least precise points-to analyses: We could not apply a more precise points-to analysis because the points-to computation itself was too costly. Other parameter models did not compute with the less

[11]JavaCard is built for applications that run on smart cards and other devices with very limited memory and processing capabilities.

[12]J2ME is a Java environment for mobile phones

[13]EPCC University of Edinburgh. The Java Grande benchmarking suite (http://www.epcc.ed.ac.uk/research/activities/java-grande/).

JRE 1.4	Battleship	JavaGrande(11)	HSQLDB
LoC	330	4.556	63.304
WALA			
(1)	11.863 (68.90%)	534.353 (65.93%)	-
(2)	32.321 (68.90%)	-	-
(3)	43.877 (68.76%)	-	-
(4)	81.780 (68.90%)	-	-
Tree			
(1)	7.619 (49.97%)	32.112.000 (61.70%)	-
(2)	7.420 (43.56%)	3.294.000 (38.72%)	-
(3)	9.180 (42.81%)	3.183.000 (38.10%)	-
(4)	12.154 (42.48%)	-	-
Graph			
(1)	8.835 (43.35%)	497.000 (46.28%)	3.808.000 (71.17%)
(2)	6.738 (43.08%)	1.871.000 (37.23%)	12.204.000 (68.70%)
(3)	8.209 (42.93%)	1.803.000 (36.78%)	-
(4)	15.704 (40.60%)	-	-

Table III
RUNTIME AND PRECISION STATISTICS FOR THE JRE 1.4 EXAMPLES.

precise points-to analyses: The parameter computation of the object tree model consumed too much time and memory and the WALA approach created too many additional parameter nodes for the subsequent summary edge computation. This shows that object graphs scale better then object trees and the WALA approach. The results of the JavaGrande programs also support this statement. They could be analyzed with the object tree and object graph model and all points-to analyses except the most precise one. However the WALA approach only finished in time with the least precise points-to analysis.

A. Runtime

Tables I, II and III show the analysis runtime including summary edges in milliseconds for each evaluated program. In general we observe huge differences in the runtime for the same program under varying options. The object graph and WALA parameter model are fast for imprecise points-to information and slow down when the information gets more precise, while the object graph is faster in most cases. Only in case of the most precise points-to analysis both perform almost equal. Object trees show the opposite behavior: They are slow on imprecise points-to information and speed up when precision increases. This effect is most visible on the larger programs like Purse and the 11 programs of the JavaGrande suite where many may-aliases slow down the computation. This effect is still present but less visible for the smaller programs. There the runtime of the phases preceding object tree computation, like call graph and points-to computation, dominates the result.

Object graphs are the clear winner in combination with the less precise points-to analyses (1), (2) and (3). They lead to the fastest SDG computation with a margin that increases with program size and decreasing points-to precision. For small programs like CorporateCard object graphs are about 5% to 15% faster than trees. The speedup increases for larger programs to about 50% and up to 98% in case of the JavaGrande programs with the least precise points-to

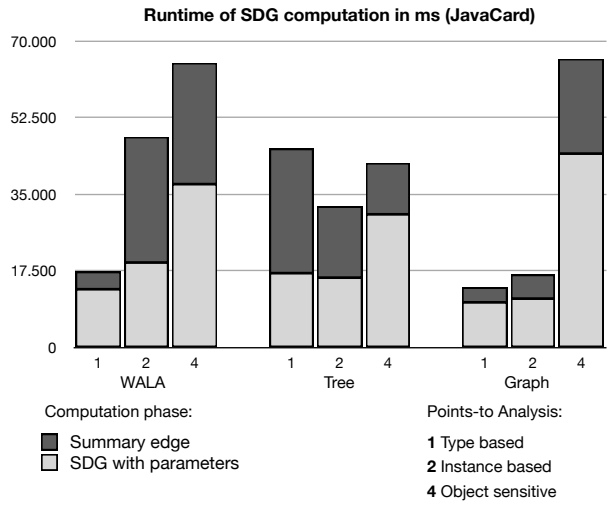

Figure 5. Average execution time of the SDG computation including summary edges for all JavaCard examples in milliseconds.

analysis (1). This trend continues to the point at which the object tree and WALA approaches take too much time and space to even finish the analysis. So object graphs allow us to analyze programs of a size that has not been possible before.

But object graphs do not perform quite as good with the most precise points-to analysis (4). This effect stems from the optimizations of the interprocedural propagation phase. In order to remove dependencies between intra- and interprocedural propagation we allow the optimized graph to contain multiple nodes for a single object field when their points-to sets differ. Object trees merge those nodes into a single parameter. This behavior harms precision a bit and results in mutual dependencies between inter- and intraprocedural propagation. But it leads to less parameters in case of a precise points-to analysis and thus a faster runtime. Object graphs may also use the unoptimized propagation phase to improve their runtime for very precise points-to analyses, but this would harm their advantage on larger programs and less precise points-to information. It is however possible to do so and should be considered whenever a very precise points-to analysis is applied.

The runtime of the parameter computation has a huge impact on the total runtime of the SDG computation, but also the structure of its result influences the successive phases: Especially the summary edge computation. Therefore we measured the runtime of the SDG computation separated from the runtime of the summary edge computation for the JavaCard examples to see the effect of the parameter model. Fig. 5 displays these results and explicitly depicts the summary edge computation time. The chart is split into 3 sections, one section for each parameter passing model. Each section shows 3 bars, and each bar denotes the result under a particular points-to analysis[14]. The fastest parameter model does not always lead to the fastest overall SDG computation, because the structure of its result can slow down the subsequent phases. One example for this behavior are the object tree SDGs of the JavaCard programs. The computation of the parameter nodes takes almost the same time for the type based (1) as for the instance based (2) points-to analysis. But due to the larger number of nodes the summary edge computation for the type based points-to analysis takes about twice as long.

B. Precision

This section focuses on the effect of the analysis options on the precision of the resulting SDG. We measure SDG precision by counting the average number of nodes contained in a context-sensitive program slice. The more precise the dependency graph is, the less nodes are contained in average in a slice. So a low average is desirable. We compute a slice for each node that corresponds to a program statement and display the precision as percentage of the average number of nodes in the slice compared to the total number of nodes. We ignore nodes that do not correspond to a statement, like parameter nodes, because their number depends on the parameter model and points-to precision. The results for each program are displayed in Table I, II and III. We also show that exception-sensitivity influences precision quite a bit.

1) Impact of parameter passing and points-to information: The impact of parameter passing and points-to information varies between the evaluated programs. The differences between the parameter models are often bigger than between the points-to analyses. We explain this effect through the differences in the refinement of field accesses that are not visible outside the scope of a method. The WALA model does not detect these field accesses at all, while the object tree model detects them implicitly during tree creation, and the object graph model runs a separate phase (Section III-C2). The importance of the refinement is especially visible at the *JavaGrande* examples. The chart in Fig. 6 contains four sections: One for the WALA parameter model, one for the object tree model and two for the object graph model. The two object graph sections show the effect of the optional refinement. The precision of the object graph SDGs without refinement changes only slightly for a more precise points-to analysis. The object graph SDGs with refinement benefit more from an increased points-to precision.

In case of the JavaCard programs the points-to analysis has little effect on the precision. However a gain of up to 4% from the imprecise type based to the more precise object sensitive points-to analysis is still visible. As the

[14]The points-to analysis no. (3) is omitted, because it contains optimizations for container classes, that are not used in the JavaCard library. Thus its results are the same as for no. (2).

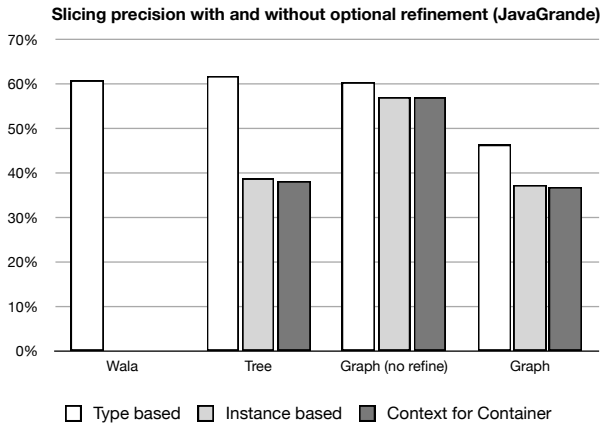

Figure 6. Average size of program slices of the examples from the JavaGrande benchmark suite.

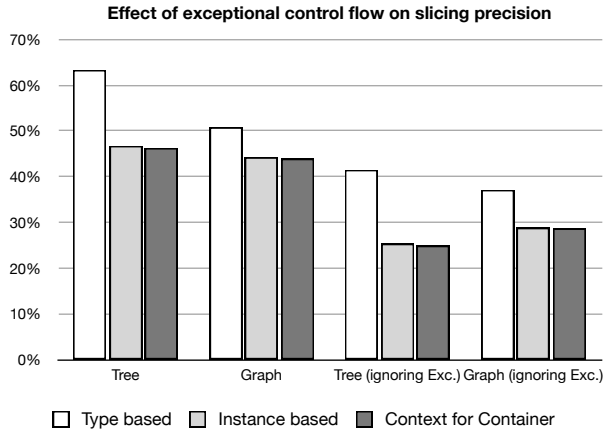

Figure 7. Average size of program slices of JavaCard, JavaGrande and Battleship examples with and without considering the effects of exceptions.

JavaCard programs are thought to run on tiny smart cards with only limited resources, they save memory and create new objects very rarely. With only few objects, the precision of the points-to analysis has no huge effect. The optional refinement of the object graph model has only little effect for the same reason.

The influence of the points-to analysis precision varies for the 3 JavaME programs. While J2MESafe and KeePassJ2ME only gain about 2% between the type and instance based analysis, Barcode gains over 11%. This shows that the expected gain in precision is hard to foresee and depends on program properties that have yet to be uncovered.

The largest program is HSQLDB. Its precision gained about 2% between type and instance based points-to analysis, while the runtime of the SDG creation increased by factor of 3.

Why does a more precise points-to analysis not always have a bigger effect on the overall precision of a SDG? In case of the JavaGrande examples the step from type based to instance based points-to analysis did have an effect, but only when we removed unreachable parameters with the refinement phase. We explain this effect with the nature of the programs in the JavaGrande benchmark suite. Those programs compute mathematical problems. The computation mostly takes place inside a particular method that creates instances of helper classes to support the computation. The references to those helper classes are not reachable outside the methods scope, and our analysis is able to detect this. The HSQLDB and J2ME examples however do not mainly run method local computations. They have a more global structure were data is passed and modified through larger parts of the program, so the effect of more precise points-to analyses and the refinement with access paths is smaller.

2) Impact of exceptional data and control flow: While the effect of the different points-to analyses has been smaller than expected, the influence of control flow and data flow through exceptions is by far greater. Our SDGs safely approximate the effect of exceptions on the behavior of the program. Currently we assume for any statement that theoretically could raise an exception that it may do so, even if this is very rarely the case in practice. We compare the precision of SDGs from the same program with and without the effects of exceptions. Fig. 7 shows the combined results of the JavaCard, JavaGrande and Battleship examples. The left two sections show the SDG precision including exceptions and the two sections on the right show the precision without control and data flow through exceptions. The average size of the program slices is almost decreased to half of their original size. This effect seems to be independent from the chosen parameter model or points-to analysis. We expect a more precise analysis of the effects of exceptions -detecting impossible exception flow- to be very beneficial for the overall precision of the SDG.

VII. Discussion

We have shown that the parameter passing model influences the precision and has a huge impact on the scalability of SDGs. The object graph model handles imprecise points-to information far better than the object tree model and it does not suffer as much from more precise information as the WALA approach. In general the optimized object graph approach delivered the best tradeoff between scalability and precision, but these optimizations should be turned off when a very precise points-to analysis is used.

The precision gained through more precise points-to analyses has been smaller than expected in most cases and varied depending on the nature and size of the program. We assume that the programs we analyzed with a more precise points-to analysis were too small and used too few objects to benefit from the precise information. We expect a significant gain through context-sensitive points-to analyses for bigger programs. Whaley and Lam showed that context-sensitive

points-to analysis is possible for larger programs [15], but then the current bottleneck remains the creation of the additional parameter nodes and the subsequent summary edge computation.

However the effects of additional control and data flow through exceptions are very visible, as the average number of nodes in the program slices almost doubled, when they where not ignored. Of course an analysis ignoring those effects is no longer sound, but this observation supports the conclusion that the current treatment of exceptions pollutes the precision of the overall result and should be investigated further.

VIII. Related work

To our knowledge only the Indus slicer [7] is -besides ours- fully implemented and can handle full Java byte code. Indus is customizable, embedded into Eclipse, and has a very nice GUI, but is less precise than our slicer e.g. in terms of interprocedural data dependences of object fields. It also computes a SDG to capture interprocedural dependencies, but it does not use parameter passing to model context-sensitive dependencies through fields. It propagates an alternative approach to data dependence computation: They essentially add data dependence between all definition statements to all matching use statements of a given field and ignore method boundaries for that purpose. Note that data dependences that cross method boundaries violate a precondition for two-phase slicing, for context-sensitive slicing in such a model one needs extra context recovery at method boundaries each time a slice is computed. Like all implementations based on SOOT [14], they do not include all possible control flow based on implicit exceptions, which may lead to unsound results.

Binkley, Harman and Krinke did an extensive evaluation of various optimization techniques for massive slicing [2]. Their work is focused on improving an already existing SDG to achieve a more precise and faster computation of program slices, rather then optimizing the generation of SDGs itself. They also did not investigate the unique features of SDGs for object oriented languages.

IX. Conclusion

Today, slicing is established as an important tool for program analyses. So it is important to improve its scalability for object oriented languages. This work takes a large step towards improved scalability of precise slicing by introducing object graphs that allow the creation of precise SDGs for large programs. We expect that the improved scalability will broaden the scope of SDG-based information flow control, and help to establish security analysis which is based on the true semantics of programs.

References

[1] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL '93*, pages 384–396, New York, NY, USA, 1993. ACM.

[2] David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.

[3] Christian Hammer and Gregor Snelting. An improved slicer for java. In *PASTE '04*, pages 17–22, New York, NY, USA, 2004. ACM.

[4] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009.

[5] John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *SAS'99*, pages 1–18, 1999.

[6] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[7] Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. Kaveri: Delivering the indus java program slicer to eclipse. In *In FASE*, pages 269–272. Springer-Verlag, 2005.

[8] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. publikation, Universität Passau, April 2003.

[9] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *ICSE '96*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[10] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[11] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, 1994.

[12] Neil Walkinshaw Marc Roper. The java system dependence graph. In *SCAM '03*, pages 5–5, 2003.

[13] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. Thin slicing. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 112–122. ACM, 2007.

[14] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[15] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*, pages 131–144, New York, NY, USA, 2004. ACM.