Data-flow-based Testing of Object-Oriented Libraries

Ramkrishna Chatterjee
Oracle Corporation
One Oracle Drive
Nashua, NH 03062, USA
ph: +1 603 897 3515
Ramkrishna.Chatterjee@oracle.com

Barbara G. Ryder
Dept. of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854, USA
ph: +1 732 445 3699
ryder@cs.rutgers.edu

Abstract

Data-flow-based testing is a well-established approach to program testing. Much object-oriented code is written as libraries; hence data-flow-based testing of object-oriented libraries is of great importance. However, finding def-use relationships in libraries written in object-oriented languages (e.g., Java and C++) is difficult because of unknown aliasing between parameters, unknown concrete types of the parameters, dynamic dispatch and exceptions. We present the first algorithm for finding def-use relationships in object-oriented libraries that overcomes the above difficulties. We also show how the information computed by our algorithm can be used in generating relevant test cases. Our algorithm is flow- and context-sensitive and based on our earlier points-to analysis [CRL99].

1 Introduction

Data-flow-based testing [RW85, FW88, OW88, LCS89, Ost90, OW91, HR94] is based on the intuition that until the result of a computation has been used during testing, a program has not been tested with respect to this computation. Many useful data-flow-based testing criteria can be defined using def-use relationships (def-uses). [RW85] showed that for a simple Pascal-like language these criteria form a hierarchy of testing criteria between all-paths criterion and all-nodes criterion. In this hierarchy a testing criterion $t_1$ subsumes another testing criterion $t_2$ if and only if coverage of $t_1$ implies coverage of $t_2$. The advantage of testing criteria based on data-flow information and other control flow graph characteristics (e.g., edges, nodes etc) is that these criteria do not depend upon any formal software specification and their satisfaction can be automatically checked (at least to a great extent). Since most software systems lack formal specification, this approach to program testing is very attractive.

In this paper, we are interested in the data-flow-based unit testing [Bei90] of O-O libraries. Since a complete program can be considered to be a library with a single entry point (i.e., main), the results of this paper also apply to complete programs. Due to exceptions, dynamic dispatch and potential aliasing at the entry node of a public interface method 1, there are new kinds of def-use relationships that need to be considered for data-flow-based testing. Consider the example library $L_{example}$ given in Figure 1. It has two public interface methods: method2 and method4. If $a_{init}$ and $a_{init2}$, the unknown objects to which $a_1$ and $a_2$ point at the entry node of method1, are the same object, statement 4 modifies the next field of $a_{init2}$ in addition to the next field of $a_{init}$. As a result, there is a def-use relationship between statements 4 and 6 if and only if $a_{init}$ and $a_{init2}$ are the same object. Consequently, even if all-path coverage is attained by a set of test cases for $L_{example}$, unless these test cases make $a_{init1}$ and $a_{init}$ identical at the entry node of the public interface method method2 (which will in turn make $a_{init}$ and $a_{init2}$ identical at call site 8), the def-use relationship between statements 4 and 6 will not be executed. This shows that due to potential def-use relationships, all-path coverage does not imply all-def-use coverage for a library. In addition to paths, the context - aliasing between the initial values at the entry nodes of public interface methods of a library and the concrete types of these initial values - also needs to be considered in choosing test data. This problem was mentioned in [HR94], but the algorithm presented there cannot compute such potential def-uses.

At a dynamically dispatched call site, the target method is chosen using the type of the receiver object. As a result, there is a new kind of p-use (use in a test expression [RW85]) of the receiver variable along every interprocedural edge from a dynamically dispatched call site to the entry node of a method that can be potentially invoked from the call site. Consider statement 5 in Figure 1. If and only if the concrete type of $a_{init}$ is $\{A, B\}$, a p-use of $r$ exists along the interprocedural edge $e_1$ from statement 5 to the entry node of $A$'s update, and consequently there is a def-use relationship between statement 3 and the edge $e_1$. Similarly, if and only if the concrete type of $a_{init}$ is $C$, a p-use of $r$ exists along the interprocedural edge $e_2$ from statement 5 to the entry node of $C$'s update, and consequently there is a def-use relationship between statement 3 and the edge $e_2$.

When an exception object is thrown by a throw statement and then later caught by a catch statement, there exists a def-use relationship between the throw statement and the entry node of the catch statement because the parameter of the catch statement is assigned the exception object. Such def-uses can also depend upon context if the exception object is an unknown initial value. For example, the exception object thrown by statement 14 (i.e., $c_{init}$) in Figure 1 is caught by the catch statement at program point 16 if and only if the concrete type of $c_{init}$ is $ET2$ or a subtype of $ET2$. As a result, there exists a def-use relationship between statement 14 and the entry node of the catch statement if and only if the concrete type of $c_{init}$ is $ET2$ or a subtype of $ET2$. Similarly, if the exception is caught by the catch
statement, there also exists a def-use relationship between statements 13 and 17. For def-use coverage, such def-uses between throws and catches and other def-uses arising from flow due to exceptions (e.g., between statements 13 and 17) also need to be exercised.

The above discussion shows that the notion of def-use relationships needs to be extended for O-O libraries. In this paper, we (1) present a new def-use algorithm that can compute the above kinds of def-use relationships (besides other, traditional def-use relationships) in libraries written in a substantial subset of Java/C++ (Sections 3, 4 and 5) and (2) show how the contexts associated with def-use relationships can be used for generating relevant test cases (Section 6). In [CRL99, Cha99] we introduced a new technique called relevant context inference for modular, flow- and context-sensitive data-flow analysis of statically typed object-oriented programming languages such as Java and C++. The def-use algorithm presented in this paper is an application of relevant context inference; it uses the points-to algorithm presented in [CRL99, Cha99] as a subroutine and shares the same overall structure.

2 Definitions

This section presents many technical definitions needed to explain the algorithm and it also delimits the subsets of Java and C++ that are handled.

RL: For the ease of presentation, in this paper we describe our algorithm for a simple object-oriented language RL that has the essential object-oriented features of Java (except threads) and C++. This allows us to simplify the presentation while demonstrating the interesting features of our algorithm. If the algorithm is understood fully for RL, then the handling of most of Java (except threads) and C++ requires handling of details but not any changes to the fundamental ideas of the algorithm. RL is defined in Figure 2. It includes single inheritance, dynamic dispatch, recursive types, exceptions and pointer assignment statements with a single level of dereferencing (any pointer assignment statement with any levels of dereferencing can be translated to this form using temporaries). The semantics of the constructs is same as in Java. RL excludes multiple inheritance, an explicit address operator (i.e., pointers to the stack), the C++ reference type, function pointers, data members of structure types (note this does not exclude data members of pointers to structure types), general pointer assignment statements, arrays (the algorithm maps array elements to a single representative element) and finally statements (as in Java).

The algorithm essentially can handle Java without threads; however, under certain circumstances we have to exclude some other features: Since in Java finalizers are invoked non-deterministically during garbage collection, we exclude finalizers that modify locations accessible outside the finalizers. We exclude static initializations that depend upon the order in which files are loaded. The algorithm can handle only those static initializations which can be safely considered (with respect to def-use analysis) to be executed in

\[\text{points-to analysis} \quad \text{[ECH94]} \text{traces value flow through variables of asserted or reference type.}\]

\[\text{Expr is a side-effect-free expression that does not have any function call. \{pattern\} means at most one occurrence of the pattern. \{pattern\}^+ \text{means zero or more occurrences of the pattern. \{a \mid b\} means a or b. The terminal symbols are underlined.}\]

\[\text{lhs = rhs, where lhs is of the form} \ p \text{or} \ p.f1 \text{and rhs is also of the form} \ q \text{or} \ q.f2.\]
program source order (or any other order specified by the user) at the start of program execution. Finally, we exclude classes whose code is constructed on the fly and not known statically.

The subset of C++ that can easily be handled by the algorithm excludes arbitrary casting, uninstantiated templates, pointers to data members and pointers to member methods (which are different from ordinary function pointers). Up-cast of a derived class to a base class and down-cast of a base class to a derived class can be handled.

**Interprocedural Control Flow Graph (ICFG):** Our algorithm operates on an ICFG [LR91]. An ICFG contains a control flow graph (CFG) for each method in the program. Each statement in a method is represented by a node in the method’s CFG and directed edges between the nodes represent control flow between the corresponding statements. Each method’s CFG has an entry node representing the entry point of the method and an exit node representing the exit point of the method. A pointer-assignment node represents an assignment statement that updates a location of pointer-assignment node.

Each method’s CFG has an entry node representing the entry point of the method and an exit node representing the exit point of the method. A pointer-assignment node represents an assignment statement that updates a location of pointer-assignment node. The test expression of an if or while statement is represented by a condition node, which has two successors: a node that represents the statement which is executed if the condition is true and another node that represents the statement which is executed if the condition is false. Each call site is represented using a pair of nodes: a call node and a return node. Information flows from a call node to the entry node of a target method and comes back from the exit node of the target method to the return node of the call node. As explained later, interprocedural edges at dynamically dispatched call sites are constructed iteratively during data-flow analysis.

**Call Graph:** A call graph contains a node for each method in the program and directed edges from callers to callees.

**Heap-name:** Like many previous researchers [LR92, EGH94, WL95, Ste96], we represent all the (potentially infinite) heap-allocated, run-time objects created at a program point n with the single name objectn, a heap-name. For example, all objects created at program point 0 in Figure 1 are represented by object0.

**Unknown initial value (UV):** varinit represents the unknown initial object (or value) to which the global or parameter var of pointer (or reference) type points at the entry node of a method. Similarly, varinit.nextinit represents the unknown initial value to which var.next points, varinit.nextinit.nextinit represents the unknown initial value to which var.next.nextpoints, and so on. For example, in Figure 1 a ainit maps to object0, while at statement 8 ainit maps to a4init. Note that the concrete value of an unknown initial value can be a subtype of the declared type of the unknown initial value, e.g., at statement 1 object0, an object of type B, maps to a4init whose declared type is A. Obviously, in the presence of recursive types the number of unknown initial values accessed in a method can be unbounded. This problem is overcome by mapping unknown initial values to a finite number of sets. All the elements in a set are represented by a single, representative name. Patterns in the access paths [Den94] of unknown initial values are used to form these sets [Cha99, CRL98].

**Interface initial value:** An unknown initial value at the entry node of a public interface method is called an in-

---

```
Program \(\Rightarrow\) \{(Class \mid Proc)\}^+
Class \(\Rightarrow\) \{Protection\} class ClassName
\{ extends\ (ClassName \mid Exception) \}
\{ \} DataMember \{ Method \}^+
DataMember \(\Rightarrow\) \{ Protection\} \{ static \} Type FieldName
Type \(\Rightarrow\) ClassName \mid PrimitiveType
PrimitiveType \(\Rightarrow\) int \mid char \mid float \mid boolean
Method \(\Rightarrow\) Protection \{ static \mid virtual \} \{ void \mid Type \}
\{ MethodName (\{Param RestParam\} \} \{ Body \}
Param \(\Rightarrow\) Type VarName
RestParam \(\Rightarrow\) Param
Proc \(\Rightarrow\) \{ void \mid Type \} ProcName (\{Param RestParam\} \}
\{ Body \}
Body \(\Rightarrow\) DeclStmt
Decl \(\Rightarrow\) Decl*
Decl \(\Rightarrow\) Type VarName
Stmt \(\Rightarrow\) AssignmentStmt \mid NewStmt \mid Call \mid If \mid While \mid ReturnStmt \mid Try \mid Throw \mid
ReturnStmt \(\Rightarrow\) return VarName
AssignmentStmt \(\Rightarrow\) Lhs \(\Rightarrow\) Rhs
Lhs \(\Rightarrow\) VarName \mid VarName.Field
Rhs \(\Rightarrow\) VarName \mid VarName.Field \mid 0 \mid Expr
Call \(\Rightarrow\) \{ VarName \Rightarrow \}
\{ VarName.MethodName.MethodName
\{ ClassName::MethodName.ProcName \}
\{ \{ VarName RestVar\} \}
\}
NewStmt \(\Rightarrow\) VarName \(\Rightarrow\) new ClassName \{ VarName RestVar\}
RestVar \(\Rightarrow\) VarName
If \(\Rightarrow\) \{ if \{ Expr \} \{ Stmt+ \} \{ else \{ Stmt+ \} \}
While \(\Rightarrow\) \{ while \{ Expr \} \{ Stmt+ \}
Throw \(\Rightarrow\) throw VarName
Try \(\Rightarrow\) \{ Try \{ Stmt+ \} \}
Catch \(\Rightarrow\) \{ catch \{ ClassName \Rightarrow VarName \{ Stmt+ \}
VarName \(\Rightarrow\) Name
FieldName \(\Rightarrow\) Name
ProcName \(\Rightarrow\) Name
MethodName \(\Rightarrow\) Name
ClassName \(\Rightarrow\) Name
Label \(\Rightarrow\) Name
Protection \(\Rightarrow\) public \mid protected \mid private
```

Figure 2: Grammar for RL
interface initial value. For example, in Figure 1 $a1_{init}$ is an interface initial value as $method2$ is a public interface method, while $a1_{init}$ is not an interface initial value as $method1$ is not a public interface method. These unknown initial values are important because their concrete values are set by a tester during unit testing of the library, while concrete values of other unknown initial values are determined by data-flow during program execution.

**Unknown initial def:** We use $\var_{initdef}$ to represent all the definitions of $\var$ reaching the entry node of a method. Here, $\var$ can be either a global or a field of an unknown initial value, and it can be of any type. For example, in Figure 1 at statement 8, $global2_{initdef}$ is associated with the entry node of $method1$, maps to all the definitions of $global2$ reaching the entry node of $method2$, while the same unknown initial def maps to definition point 6 at statement 11. Intuitively, unknown initial defs and unknown initial values help in efficiently representing all the contexts in which a method can be invoked (and that are relevant for def-use analysis) and thus, facilitate parametric analysis of libraries without knowing the contexts in which the library methods can be invoked.

**Interface initial def:** An unknown initial def at the entry node of a public interface method is called an interface initial def.

**Compatible classes:** Two classes are compatible if and only if they are the same class or they are related by a subtype-supertype relationship. In Figure 1, class $A$ is compatible with classes $B$ and $C$, while classes $B$ and $C$ are incompatible.

**Compatible unknown initial values:** Two unknown initial values are compatible if and only if their declared classes are compatible.

**Points-to:** A points-to is a pair of the form: $\langle \var, \obj \rangle$, where $\var$ is a local pointer variable, a global pointer variable, an unknown initial value's field of pointer type or a heap-name's field of pointer type; and $\obj$ is an unknown initial value or heap-name. $\obj$ can also be $null$, which is treated as a special heap-name. For example, in Figure 1 the points-to $\langle a1_{init}.next, a3_{init} \rangle$ holds at the bottom of statement 4.

**Def-Use Relationship:** A def-use relationship is a triplet $\langle \loc, \text{def-point}, \text{use-point} \rangle$, where $\loc$ is

1. a user-defined variable,
2. a field of an unknown initial value or
3. a field of a heap-name

of any type, and there exists an execution path $p$ from the entry node of a public interface method to $\text{use-point}$ and an environment $e$ at the entry node of the public interface method, such that if $p$ is executed starting with the environment $e$ at the entry node of the public interface method, $\loc$ is defined at $\text{def-point}$ and this definition of $\loc$ is used at $use-point$. In Figure 1, $\langle global3, 13, 17 \rangle$ represents the def-use relationship due to the definition and use of $global3$ at statements 13 and 17 respectively.

### 3 Overview of def-use algorithm

First we will give a brief overview of the various steps of the def-use algorithm using the example in Figure 1 for illustration. Then we will present the details of these steps. Due to limited space we will only summarize the essential features of the points-to algorithm.

The def-use algorithm (Def-Use-Algo) is an iterative worklist algorithm that is flow- and context-sensitive [LR92]; its data-flow computation is affected by the ordering of statements along intraprocedural paths and it restricts (approximately) data-flow only to interprocedural paths with matching calls and returns [SP81]. Def-Use-Algo takes as input a statement-level interprocedural control flow graph ([ICFG] [LR92]). From this ICFG an initial, approximate call graph is formed and then decomposed into strongly connected components (SCC’s). The rest of the Def-Use-Algo consists of three major phases that are performed using the SCC condensation (SCC-DAG). Each of the first two phases propagate points-to and def-use information respectively in two successive subphases.

#### 3.1 Phase 0

In this phase Def-Use-Algo constructs a safe overestimate of the call graph called the initial call graph by resolving dynamically dispatched calls using hierarchy analysis [DMM96]. Then Def-Use-Algo uses a linear-time algorithm [CLR92] to construct the SCC-DAG of the initial call graph. Note that the initial call graph need not be precise, it only needs to be a safe overestimate; the precision of any safe initial call graph only affects the efficiency of Def-Use-Algo, and not the safety of the computed solution. The initial call graph can be made more precise (e.g., by using rapid type analysis [BS96]); however, in practice we have found hierarchy analysis to be adequate. For $L_{example}$ each SCC contains exactly one method and $\{A.update, C.update, Lib.method1, Lib.method2, Lib.method3, Lib.method4\}$ is a listing of the SCC’s. The following three phases are performed using the SCC condensation:

#### 3.2 Phase 1

In this phase Def-Use-Algo traverses the SCC-DAG in a reverse topological order (bottom-up) and analyzes each method assuming parameters and global variables have unknown initial values. $\{A.update, C.update, Lib.method1, Lib.method2, Lib.method3, Lib.method4\}$ is a reverse topological listing of the SCC’s of $L_{example}$. It performs the following two subphases on each SCC:

##### 3.2.1 l-pta

During this first subphase Def-Use-Algo performs points-to analysis and this subphase is same as the Phase I of the points-to algorithm described in [CRL99, CRL98]. For each method Def-Use-Algo computes, in terms of unknown initial values, a safe approximation to the method’s complete transfer function for points-to analysis. We will call this approximation the **pointer summary transfer function**. This function summarizes the possible effects of method invocation on values of pointers. Intuitively, it is a safe approximation in the sense that given the values of pointer locations at a call site of the method, the computed set of values of any pointer location after the call, contains all values possible during execution plus perhaps some spurious values. The concrete representation of this function for method $M$ is the set of data-flow elements representing values of nonlocal pointers that reach the exit node of $M$. The

---

*We refer to pointer and reference types interchangeably in this paper.*
pointer summary transfer functions of methods in the same SCC have cyclic dependencies, so they are computed simultaneously by fixed point iteration. In contrast, the pointer summary transfer functions of methods in different SCC’s have hierarchical dependencies (or no dependence at all), and hence are computed by bottom-up traversal of SCC-DAG without iteration.

The results of this subphase on a SCC are two-fold: (i) a points-to solution at each node of every method in the SCC and (ii) a pointer summary transfer function for every method in the SCC, both of which are parametrized by unknown initial values and conditions on unknown initial values.

**Conditional contexts:** The effects of a method on pointers is calculated dependent on conditions on (1) the aliasing between unknown initial values of parameters and globals, and (2) the concrete types (run-time types) of these unknown initial values. For example, statement 4 in Figure 1 can modify the next field of a2init if and only if a1 and a2 are aliased at the entry of method1, and hence l-pta infers that on the top of statement 5, a2init.next points to a3init under the condition that a1init and a2init are the same object at the entry of method1. Similarly, statement 5 can invoke A.update or C.update depending upon the concrete type of a1init, and hence l-pta infers that on the top of statement 6, global1 points to object1 under the condition that the concrete type of a1init ∈ {A,B}, and global1 points to object2 under the condition that the concrete type of a1init ∈ {C}. These conditions are evaluated at a call site of a method by instantiating the unknown initial values with their values implied by actual-to-formal bindings at the call site, to propagate data-flow elements from the exit node of the callee to the return node of the call site in a context-sensitive manner. For example at call site 11, a1init maps to object9 and a2init maps to object10. As a result, the condition saying a1init is same as a2init evaluates to false and the data-flow elements that are conditioned on this condition are not propagated from the exit node of method1 to statement 12. This means at statement 12 object10.next points to object10 and object10.next does not point to object10, instead object10.next points to null (i.e., object10.next retains its value across the call).

**Relevant contexts:** Rather than calculate all possible conditions, Def-Use-Algo calculates only those conditions which may affect points-to (or def-use) information, by inferring them from the code of the method and those other methods it may invoke directly or indirectly during its lifetime. Care is taken to observe those object fields actually used by this method directly or indirectly through calls; conditions are inferred for only those fields which are used in this sense. For example, method1 in Figure 1 uses the next fields of a1init and a2init, but it does not use the next field of a3init. As a result, although statement 4 can modify a3init.next as well as a2init.next due to aliasing of a1init with a2init and a3type, only a data-flow element representing conditional modification of a2init.next is generated (as explained earlier). This is what makes Def-Use-Algo feasible: the representation of a summary transfer function is in terms of relevant contexts only, while it can be safely used across all possible contexts (which could be exponential, even infinite).

### 3.2.2 I-dua

During this second subphase Def-Use-Algo computes def-use relationships using the points-to solution computed in I-pta. For each method Def-Use-Algo computes, in terms of unknown initial values and unknown initial defs, a safe approximation to the method’s complete transfer function for def-use analysis. We will call this approximation the **def-use summary transfer function**. Intuitively, this function summarizes the possible effects of method invocation on variable definitions. Again, it is a safe approximation in the sense that given the context at a call site of the method, the computed set of definitions of any location after the call, contains all definitions possible during execution plus perhaps some spurious ones. The def-use summary transfer function of a method M is the set of data-flow elements representing definitions of nonlocal locations that reach the exit node of M. Again, the def-use summary transfer functions of methods in the same SCC have cyclic dependencies, so they are computed simultaneously by fixed point iteration. In contrast, the def-use summary transfer functions of methods in different SCC’s have hierarchical dependencies (or no dependence at all), and hence are computed by bottom-up traversal of SCC-DAG, without iteration. The above two subphases could be interleaved as long as I-pta occurs on a SCC before the corresponding I-dua.

The results of this subphase on a SCC are two-fold: (i) a def-use solution at each node of every method in the SCC and (ii) a def-use summary transfer function for every method in the SCC, both of which are parametrized by unknown initial values, unknown initial defs and conditions on unknown initial values.

**Conditional contexts:** Similar to points-to’s, the def-use relationships are also calculated dependent on conditions on (1) the aliasing between unknown initial values of parameters and globals, and (2) the concrete types of these unknown initial values. For example, in Figure 1 the def-use relationship between the statements 4 and 6 due the definition of a2init.next at statement 4 and later its use at statement 6 is computed dependent on the condition that a1init and a2init are the same object at the entry of method1. Similarly, the def-use relationship between the statements 1 and 7 due to the definition and use of global1 is computed dependent on the condition that the concrete type of a1init ∈ {A,B} and the def-use relationship between the statements 2 and 7 due to the definition and use of global1 is computed dependent on the condition that the concrete type of a1init ∈ {C}. Again, unknown initial values and unknown initial defs are instantiated at a call site of a method with their values implied by actual-to-formal bindings at the call site, to propagate data-flow elements from the exit node of the callee to the return node of the call site in a context-sensitive manner. For example, at call site 11 a1init maps to object9. As a result, the condition that the concrete type of a1init ∈ {A,B} is true and the definition of global1 at statement 1 is propagated to the return node of call site 11. In contrast, the definition of global1 at statement 2 is not propagated to the return node as the associated condition is false at the call site.

**Unknown initial def:** When a global or a field of an unknown initial value is used at a statement in a method and there exists a definition clear path from the entry node of the method to the statement, the definition point of the corresponding def-use relationship at the statement is parametrized by the unknown initial def of the global or the field of the unknown initial value at the method entry node. For example, in Figure 1 the initial value of a2init.next is used at statement 6 if a1 and a2 are not aliased at the entry node of method1, and hence the definition point of the def-use relationship due to the definition and use of a2init.next,
computed at statement 6, is parametrized by the unknown initial def of \( a_{init.next} \) (i.e., \( a_{init.next.initdef} \)).

### 3.3 Phase II

In this phase Def-Use-Algo traverses the SCC-DAG in a topological order (top-down) and performs the two sub-phases II-pta and II-dua on each SCC in order. During both the subphases, the propagation within a SCC is done iteratively until a fixed point is reached, while propagation across SCC’s is done in a top-down manner without iteration. Phase II involves only the entry nodes and call nodes, and consequently it is likely to be quite efficient in practice. The empirical results for II-pta have been published in [CRL99].

**II-pta:** In this first subphase Def-Use-Algo propagates the concrete values of unknown initial values to the entry nodes of methods. This is same as Phase II of the points-to algorithm presented in [CRL99, CRL98]. For example, in Figure 1 II-pta propagates \( object_b \) and \( object_{c2} \) from call site 11 to the entry node of \( method_1 \) as a concrete values of \( a_{init} \) and \( a_{init} \) respectively. II-pta treats an interface initial value like a concrete value and propagates it to the entry nodes of other methods if, through a defining binding at a call site, the interface initial value is the value of an unknown initial value at the entry of a target. For example, at call site 8 in Figure 1, this subphase propagates the interface initial value \( a_{init} \) to the entry of \( method_1 \) as a concrete value of \( a_{init} \).

**II-dua:** In this second subphase Def-Use-Algo propagates concrete values of unknown initial defs to the entry nodes of methods. For example, at statement 10 in Figure 1, the next field of \( object_{c10} \) is initialized (defined), and hence at call site 11, II-dua propagates program point 10 to the entry node of \( method_1 \) as a concrete value of \( a_{init.next.initdef} \), the unknown initial def of \( a_{init} \). Again, II-dua treats an interface initial def like a concrete value and propagates it to the entry nodes of other methods if, through a defining binding at a call site, the interface initial def is the value of an unknown initial def at the entry of a target. For example, at call site 8 II-dua propagates the interface initial def \( a_{init.next.initdef} \) to the entry node of \( method_1 \) as a concrete value of the unknown initial def \( a_{init.next.initdef} \).

### 3.4 Phase III

This phase involves only non-entry nodes at which the unknown initial values and unknown initial defs in the parametrized def-use solution computed during I-dua are instantiated by their concrete values computed in Phase II. After this phase, the def-use solution at each node is expressed entirely in terms of program variables, heap-names, fields of heap-names, interface initial values, fields of interface initial values, definition points, interface initial defs and use points.

Those instantiations of the unknown initial values in parametrized def-uses for which the conditions associated with the def-uses evaluate to true, yield the final def-use solution. When a condition involves an interface initial value, conservative, worst-case assumptions are made about the interface initial value to evaluate the condition. Note that a potential non-alias (a condition that asserts the inequality of two unknown initial values) evaluates to false only if both operands of the potential non-alias are instantiated by the same interface initial value, even if both operands of a potential non-alias are instantiated by the same heap-name, the potential non-alias evaluates to true because the heap-name can represent more than one run-time object.

For example, in Figure 1 during I-dua the def-use relationship between the statements 2 and 7 due to the definition and use of \( global_1 \) is computed dependent on the condition that the concrete type of \( a_{init} \in \{C \} \). The interface initial value \( a_{init} \) (bound at call site 8) and \( object_{c2} \) (bound at call site 11) are the concrete values of \( a_{init} \) computed during I-dua. When \( a_{init} \) is instantiated with \( object_{c2} \) during Phase III, the condition asserting concrete type of \( a_{init} \in \{C \} \) evaluates to false. On the other hand, when \( a_{init} \) is instantiated with \( a_{init} \) during Phase III, the same condition evaluates to true because Def-Use-Algo makes conservative, worst-case assumptions about the interface initial value \( a_{init} \). Hence the final def-use solution contains the def-use relationship resulting from the definition and use of \( global_1 \) at statements 2 and 7 respectively, and this def-use is conditioned on the fact that the concrete type of the interface initial value \( a_{init} \in \{C \} \). Similarly, during I-dua the def-use relationship between the statements 4 and 6 due the definition of \( a_{init.next} \) at statement 4 and later its use at statement 6 is computed dependent on the condition that \( a_{init} \) and \( a_{init} \) are the same object at the entry of \( method_1 \). In addition to the concrete values of \( a_{init} \) mentioned above, II-dua computes the interface initial value \( a_{init.next} \) (bound at call sites 2 and \( object_{c2} \) (bound at call site 11) as the concrete values of \( a_{init} \). When \( a_{init} \) and \( a_{init} \) are instantiated with their concrete values in Phase III, the condition that \( a_{init} \) and \( a_{init} \) are the same object can evaluate to true only for the case in which \( a_{init} \) is instantiated with \( a_{init} \) and \( a_{init} \) is instantiated with \( a_{init} \) (recall that Def-Use-Algo makes conservative, worst-case assumption that the interface initial values \( a_{init} \) and \( a_{init} \) are possibly aliased).

For all other instantiations the condition evaluates to false. The condition asserting the equality of an interface initial value (or unknown initial value) with a heap-name evaluates to false because for a specific invocation of a public interface method (or method), the interface initial value (or unknown initial value) represents an object instance allocated before the call, while the heap-name represents an object instance allocated during the lifetime of the call. Thus, the final def-use solution contains the def-use relationship resulting from the definition and use of \( a_{init.next} \) at statements 4 and 6 respectively, and this def-use is conditioned on the fact that \( a_{init} \) and \( a_{init} \) are the same object at the entry node of the public interface method \( method_2 \).

Phase III is completely demand-driven and is performed only at those nodes whose final solution is needed.

### 3.5 Modularity

As explained in [CRL99] relevant context inference is a modular data-flow analysis technique. Since Def-Use-Algo is an application of relevant context inference, Def-Use-Algo also has the same characteristic. During the construction of the initial call graph using hierarchy analysis, each method needs to be in memory once. After this, each node of SCC-DAG (and hence each method) needs to be in memory only three more times, once during each of the phases I, II and III. The rest of the time, only a method’s pointer summary transfer function (during I-pta) or def-use summary transfer function (during I-dua), or the Phase II solution at the entry node of a method needs to be in memory. Hence, this is a modular approach and requires less memory than other whole-program-analysis techniques, in which a method cannot be moved out of memory without the possibility of it
There are two kinds of PTA-dfelm:

- dfelm-a. [ECFInfo, relevant context, points-to]
- dfelm-b. [relevant context, exception object]

There are five kinds of DUA-dfelm:

- dfelm-c. [ECFInfo, relevant context-1, (var, def-point, (relevant context-2, value))]
- dfelm-d. [relevant context, exception object, throw-point]
- dfelm-e. [relevant context, (var, def-point, use-point)]
- dfelm-f. [exception object, throw-point, catch-point]
- dfelm-g. [relevant context, (var, def-point, use-edge)]

Figure 3: Phase I data-flow elements

being needed again, until the final solution is computed. In these techniques, if the whole program is not kept in memory, there is no a priori constant bound on the number of times a method needs to be moved into or out of memory.

As discussed in [CRL99], in the worst case the entire initial call graph may be a single SCC and Def-Use-Algo may need to keep the whole program in memory. However, empirical results in [CRL99] for C++ programs show that SCC’s are quite small in practice and Def-Use-Algo is able to analyze almost a method at a time. In some very specific cases, the SCC’s may be occasionally large, however, even in these cases the entire initial call graph is unlikely to be a single SCC. For example, in parsing, the SCC’s of methods dealing with statements are likely to be different from the SCC’s of methods dealing with types.

Phase I data-flow elements

Def-Use-Algo computes two kinds of data-flow elements during phase I: PTA-dfelm and DUA-dfelm. These are shown in Figure 3.

PTA-dfelm are used for points-to analysis and are computed during subphase I-pta. PTA-dfelm represent (a) values of pointer variables and (b) exception objects. dfelm-a are used for propagating values of locations of pointer type. dfelm-b are used for propagating exception objects from throw statements to corresponding catch statements (if any). The propagation of exception objects requires data-flow elements of a separate kind because exception objects are propagated through implicit stack locations until they are caught by a catch statement, whereupon they are assigned to the parameter of the catch statement and can be propagated using dfelm-a.

DUA-dfelm are used for def-use analysis and are computed during subphase I-dua. DUA-dfelm represent variable definitions and def-use relationships. DUA-dfelm are computed using PTA-dfelm because computation of def-use relationships require the points-to solution. dfelm-e, dfelm-f and dfelm-g are needed because they represent three different kinds of def-use relationships described in Section 1. dfelm-c are needed for propagating variable definitions to their use points, where they are used for computing dfelm-e and dfelm-g depending upon whether the use is in a non-test expression or in a test expression. dfelm-d are needed for propagating exception objects from throw statements to catch statements, where they are used for computing dfelm-f. dfelm-b cannot be used for this purpose because they do not encode the throw point in them.

We will use dfelm to denote both PTA-dfelm and DUA-dfelm.

4.1 Data-flow elements for points-to analysis

dfelm-a: A dfelm-a represents a value of a location of pointer/reference type. Intuitively, an ECFInfo or exception context-flow information stores the signature of an uncaught exception (if any) along paths through which a PTA-dfelm reaches a program point from the entry node of the method containing the program point, and it determines the future propagation of the PTA-dfelm or DUA-dfelm from the program point. An ECFInfo is one of the following:

- except-type, the run-time type of the exception object,
- iv, when the unknown initial value iv is thrown as exception object or
- empty, for propagation along an exception-free path from the entry node of a method to a statement in the method.

For example, statement 14 in Figure 1 throws e1init and hence when the dfelm-a representing the value of global3 is propagated across statement 14, e1init becomes the ECFInfo of the new dfelm-a propagated to statement 15. The reason for storing an unknown initial value uv as the signature of the exception when uv is thrown as the exception object is that the concrete type of the exception can be the declared type of uv or any subtype of the declared type. The concrete type of the exception will be determined using concrete values of uv. For example, in Figure 1 when methodd3 is invoked from statement 20, the concrete type of the exception thrown at statement 14 is ET4, and thus on returning from the call, the ECFInfo e1init is replaced by ET4 and the new dfelm-a is propagated to statement 21. The reason why singleton ECFInfo is sufficient is that in the absence of finally statements (as in RL) exceptions do not stack up.

A relevant context has the form:

(alias context, type context, exception context).

An alias context is empty or it is a conjunction of potential aliases and potential non-aliases between unknown initial values. Each potential alias has the form:

\((\textit{uv}v_1 \textit{eq} \textit{uv}v_2)\)

and each potential non-alias has the form:

\((\textit{uv}v_1 \textit{neq} \textit{uv}v_2)\),

where \(\textit{uv}v_1\) and \(\textit{uv}v_2\) are unknown initial values. For example, in Figure 1, at the bottom of statement 4 the points-to \(\langle a_2\textit{init}.\textit{next}, a_3\textit{init} \rangle\) is conditioned on the potential alias \((a_1\textit{init} \textit{eq} a_2\textit{init})\), while the the points-to \(\langle a_1\textit{init}.\textit{next}, a_2\textit{init} \rangle\) is computed conditioned on the potential non-alias \((a_1\textit{init} \textit{neq} a_2\textit{init})\).

A type context is empty or it is a conjunction of type constraints. A type constraint in a type context is inferred when an unknown initial value is the receiver of a dynamic dispatch. Each type constraint has the form:

\((\text{type}(\textit{uv}) \in T:x)\).
where \( uiv \) is an unknown initial value. \( T \) is a class and \( x \) is a dynamically dispatched method defined in \( T \) (a virtual method in \( C^{++} \)). \( T::x \) represents the set of classes containing \( T \) and all the subtypes of \( T \) for which a virtual invocation of method \( x \) will be resolved to the definition of method \( x \) in class \( T \), i.e., \( T::x \). The constraint means that the associated \( dfelm \) is valid only in those contexts in which the concrete, run-time type of \( uiv \) (not the declared type) belongs to \( T::x \). For example, in Figure 1 \( A::update \) represents \( \{ A,B \} \); and at the bottom of statement 5, the points-to \((global1, object1)\) is computed conditioned on the type constraint \((type(a1 init) \in A::update)\), while the points-to \((global1, object2)\) is computed conditioned on the type constraint \((type(a1 init) \in C::update)\).

An exception context is empty or it is a conjunction of type constraints. The type constraints in an exception context are inferred when an unknown initial value is propagated as an exception object. Each type constraint in an exception context has one of the following two forms:

1. \((\text{type}(uiv) \leq T)\)
2. \((\text{type}(uiv) \not\leq T)\)

The first type constraint says that the associated \( dfelm \) holds only in those contexts where the concrete type of the unknown initial value \( uiv \) is class \( T \) or a subtype of \( T \); while the second type constraint says that the the associated \( dfelm \) holds only in those contexts where the concrete type of the unknown initial value \( uiv \) is neither \( T \) nor a subtype of \( T \). For example, in Figure 1, at the bottom of statement 17 the points-to \((u1, object1)\) is computed conditioned on the exception context \((\text{type} (a1 init) \leq ET2)\), because the points-to \((global3, object1)\) is propagated from statement 14 to statement 17 when the exception object \((e1 init)\) thrown at statement 14 is caught at statement 16, and this is possible if and only if \((\text{type}(a1 init) \leq ET2)\). In contrast, at statement 18 the points-to \((u1, null)\) is computed conditioned on the exception context \((\text{type}(a1 init) \not\leq ET2)\), because the points-to \((u1, null)\) is propagated from statement 14 to statement 18 when the exception object \((e1 init)\) thrown at statement 14 is not caught at statement 16, and this is possible if and only if \((\text{type}(a1 init) \not\leq ET2)\).

These relevant contexts are inferred by Def-Use-Algo during data-flow analysis and they summarize the contexts under which the corresponding \( dfelms \) hold. When a \( dfelm \) \( dfe \) is propagated from the exit node of a method to a call site that invokes the method, one of the following three things happens for each conjunct of the relevant context of \( dfe \): the conjunct evaluates to \( true \), it evaluates to \( false \) (\( dfe \) is not propagated to the call site in this case), or it is translated into a similar conjunct involving the unknown initial values of the caller. For example, when the \( dfelm-a \) \([\text{empty}, (a1 init eq a2 init), empty, empty, (a2 init, next, a3 init)]\) is propagated from the exit node of method1 to the return node of call site 8, it is translated into \([\text{empty}, (a5 init eq a5 init), empty, empty, (a5 init, next, a6 init)]\); while the same \( dfelm-a \) is not propagated from the exit node of method1 to the return node of call site 11 because the potential alias \((a1 init eq a2 init)\) translates to \((object9 eq object10)\), which is false.

We will use \( \text{Empty-context} \) to represent the relevant context \([\text{empty}, empty, empty]\).

**dfelm-b**: A \( dfelm-b \) is used for propagating an exception object. \( \text{exception object} \) is either an unknown initial value or a heap-name. Intuitively, \( dfelms-b \) are needed because they determine the values of the parameters of the catch statements. An \( \text{exception object} \) is assigned to the parameter of a catch statement that catches the exception object. For example, in Figure 1 the exception object thrown at statement 14 is propagated as the \( dfelm-b \) \([\text{Empty-context}, e1 init]\) to statement 15, the exit of the try statement. From statement 15, the \( dfelm-b \) \([\text{empty}, empty, (type(e1 init) \leq ET2)\. e1 init]\) is propagated to statement 16 because the exception is caught by the catch statement under the type constraint \((type(e1 init) \leq ET2)\). At statement 16, the above \( dfelm-b \) determines the value of \( exp \), the parameter of the catch statement. Similarly, from statement 15 the \( dfelm-b \) \([\text{empty}, empty, (type(e1 init) \not\leq ET2)\. e1 init]\) is propagated to statement 18 because the exception is not caught by the catch statement under the type constraint \((type(e1 init) \not\leq ET2)\). From statement 18 this \( dfelm-b \) is propagated to call sites of method3.

### 4.2 Data-flow elements for def-use analysis

There are five different kinds of data-flow elements which represent different kinds of def-use information needed during phase I-dua. These data-flow elements are computed during I-dua using PTA-\( dfelms \) computed during I-pfa.

**dfelm-c**: \( dfelms-c \) are used for propagating variable definitions to their use points. Each \( dfelm-c \) has the form \([\text{ECFINfo, relevant context-1, [var, def-point, (relevant context-2, value)]]}\), where (1) \( var \) is a user defined variable, a field of an unknown initial value or a field of a heap-name of any type and (2) \( def-point \) is a program point (i.e., a definition point) or an unknown initial def. relevant context-1 and relevant context-2 are relevant contexts as defined in the case of PTA-\( dfelms \). relevant context-1 represents the contexts in which \( var \) is defined at \( def-point \). If \( var \) is of pointer type, \( value \) is the object that was assigned to \( var \) at \( def-point \) and relevant context-2 represents the contexts in which \( var \) takes this value. If \( var \) is not a pointer, relevant context-2 is \( \text{Empty-context} \) and value is \( don't-care \). As we will see later, value is needed for computing \( p\)-uses on interprocedural edges at dynamically dispatched call sites. For example, \([\text{empty}, (a1 init eq a2 init), empty, empty, (a2 init, next, 4, (\text{Empty-context}, a3 init))]\) represents the potential definition of \( a2 init, next \) at statement 4 in Figure 1. In this case, relevant context-1 is \((a1 init eq a2 init), empty, empty\) because \( a2 init, next \) (the var) is defined at statement 4 under this context; in contrast, relevant context-2 is \( \text{Empty-context} \) because the right-hand-side of the assignment statement \((a3)\) has the value \( a3 init \) in all contexts. Similarly, \([\text{empty}, \text{Empty-context}, (global2, 6, ((a1 init eq a2 init), empty, empty, (a3 init))]\) represents the definition of \( global2 \) at statement 6 in Figure 1. In this case, relevant context-1 is \( \text{Empty-context} \) because \( global2 \) (the var) is defined at statement 6 in all contexts, and relevant context-2 is \((a1 init eq a2 init), empty, empty\) because the right-hand-side of the assignment statement \((a3 init, next)\) has the value \( a3 init \) under this context.

**dfelm-d**: \( dfelms-d \) are used for propagating exception definitions from throw statements to catch statements which catch the exceptions, and thus facilitate computation def-use relationships between throw statements and corresponding catch statements. A \( dfelm-d \) has the form \([\text{relevant context, exception object, throw-point}]\), where (1) \( throw-point \) is the program point at which exception object has been thrown and (2) \( exception object \) is an unknown initial value or a heap-name which represents the thrown exception object. relevant context represents the contexts in which exception object is thrown at throw-point. For example, \([\text{Empty-
context, 1init, I] represents the exception thrown at statement 14 in Figure 1. Here relevant context is Empty-context because e1 points to e1init at statement 14 (and hence e1init is the exception object) for all calling contexts of method3.

dfelm-e: dfelms-e capture def-uses of variables and object fields, although these may be parameterized in terms of unknown initial def s and unknown initial values. Each dfelm-e has the form [relevant context, (var, def-point, use-point)], where (1) var and (2) def-point are the same as in dfelm-c, and (3) use-point is a program point. Intuitively, var is a location which is defined at def-point in contexts represented by relevant context and this definition of var is used at use-point. For example, the dfelm-e [(\langle a1init \text{ eq } a2init \rangle, empty, empty), \langle a2init, next, 4, 6 \rangle] represents the def-use of a2init.next at statements 4 and 6 in Figure 1. The relevant context is \langle a1init \text{ eq } a2init \rangle because a2init.next is defined at statement 4 only in those calling contexts of method1 in which a1 and a2 are aliased. Similarly, the dfelm-e [(\langle a1init \text{ neq } a2init \rangle, empty, empty), \langle a2init.next, a2init.next, initdf, 6 \rangle] represents the def-use of a2init.next because of the definitions of a2init.next reaching statement 6 from the entry of method1. The relevant context is \langle a1init \text{ neq } a2init \rangle because there exists a definition-free path for a2init.next from the entry node of method1 to statement 6 only in those calling contexts of method1 in which a1 and a2 are not aliased.

dfelm-f: A dfelm-f represents a def-use relationship between a throw statement and a catch statement that catches the exception thrown at the throw statement. Each dfelm-f has the form [relevant context, (exception-object, throw-point, catch-point)], where (1) exception-object is an unknown initial value or a heap-name, (2) throw-point is the number of a throw statement and (3) catch-point is the number of the entry node of a catch statement. It means exception-object thrown at throw-point in calling contexts represented by relevant context is caught at catch-point. For example, in Figure 1 [(empty, empty, (\text{type}(e1init) \leq ET2)), \langle e1init, initdf, 14, 16 \rangle] represents the def-use relationship between the throw statement 14 and the catch statement 16 because the exception object e1init thrown at statement 14 is caught at statement 16 only in those calling contexts of method3 in which the concrete type of e1init is ET2 or a subtype of ET2.

dfelm-g: These DUA-dfelm-s represent p-uses (i.e., use in a test expression). Each DUA-dfelm has the form [relevant context, (var, def-point, use-edge)], where (1) use-edge is an edge out of a condition node (representing the test expression of an if or while) or it is an interprocedural edge between a dynamically dispatched call site and one of its targets, and (2) var and (3) def-point are same as in dfelm-c. For example, [(\text{empty}, \text{type}(a1init) \in A::\text{update}), empty, (r, 3, (5, A::\text{update}))] and [(\text{empty}, \text{type}(a1init) \in C::\text{update}), empty, (r, 3, (5, C::\text{update})] represent the p-uses due to dynamic dispatch at statement 5 in Figure 1. This is because in those calling contexts of method1 in which (\text{type}(a1init) \in A::\text{update}), the value of the receiver variable r at statement 5 is defined at statement 3 and the target of the dynamic dispatch is A::update, while in those calling contexts of method1 in which (\text{type}(a1init) \in C::\text{update}), the value of the receiver variable r at statement 5 is defined at statement 3 and the target of the dynamic dispatch is C::update.

In Appendix A we present the def-use-analysis transfer function for each kind of ICFG node, and illustrate the different steps of Def-Use-Algo using the example in Figure 1.

Limiting relevant context: Theoretically, the number of conjuncts in the relevant context of a dfelm could be unbounded. Thus, if all the conjuncts in the relevant context of a dfelm needed to be maintained for safety, Def-Use-Algo would be infeasible. However, as the following theorem shows, safety can be ensured by maintaining only a subset of the conjuncts.

Let d be a dfelm at the exit node of a method M, C be a call site that invokes M and rc be the relevant context of d. If rc evaluates to true at C, any relevant context t that is contained in rc (i.e., the set of conjuncts of t is a subset of the set of conjuncts of rc) also evaluates to true at C. As a result, we have the following theorem which is a generalization of the theorem in [CRL99] (the theorem in [CRL99] is only for PTA-dfelm-s, while the following theorem is for both PTA-dfelm-s and DUA-dfelm-s):

**Theorem 1** For any dfelm with relevant context r, it is safe to replace r with a relevant context s that is contained in r.

Due to Theorem 1, instead of the complete relevant context, Def-Use-Algo can store any subset of these conjuncts without compromising safety, although this may cause propagation of spurious dfelms to call sites where only a part of the original relevant context is valid (i.e., we are using approximate context sensitivity). At present we use a simple heuristic: if the user specifies a bound of k on the number of conjuncts of a specific kind, we store the first k conjuncts of this kind associated with a dfelm, the rest of the conjuncts are dropped.

5 Computation of the final def-use solution

In phase III unknown initial values and unknown initial defs in dfelms-e, dfelms-f and dfelms-g are instantiated by their concrete values computed in Phase II. Recall that an interface initial value can be the concrete value of an unknown initial value and an interface initial def can be the concrete value of an unknown initial def. Unknown initial values are instantiated by their concrete values computed in II-pta, while unknown initial defs are instantiated by their concrete values computed in II-dua. Those instantiations of DUA-dfelm-s for which relevant contexts either (1) evaluate to true or (2) they are instantiated into relevant contexts involving only the interface initial values of a single public interface method and these instantiated relevant contexts can evaluate to true by making conservative, worst-case assumptions about these interface initial values, yield the final def-use solution.

Each element of the final def-use solution at a node n has one of the following three forms:

1. \[\langle rc, \langle var, def-point, use-point \rangle \rangle, \text{where def-point is a program point or an interface initial def, use-point is n and var is a global variable, local variable, field of a heap-name or field of an interface initial value. Elements of this form express def-uses of assignable memory locations.}\]

2. \[\langle rc, \langle excp-object, throw-point, catch-point \rangle \rangle, \text{where throw-point is a throw statement number, catch-point is n and excp-object is a heap-name or an interface initial value. Elements of this form express def-uses between throw and catch statements.}\]

3. \[\langle rc, \langle var, def-point, use-edge \rangle \rangle, \text{where either n is a condition node and use-edge is a decision edge out of n or}\]
is part of the relevant context of any def-use relationship. If contexts of def-use relationships computed during Phase III

6 Algorithm for finding reduced context cover

In all cases rc is either Empty-context (meaning the relevant context before the instantiation evaluated to true after the instantiation), or it is a relevant context involving only the interface initial values of a single public interface method and it can evaluate to true by making conservative, worst-case assumptions about these interface initial values.

For example, consider the public interface method method2 in Figure 1. At program point 7, the def-use relationship

\[ \langle \text{hasType}(\text{method}2), \langle \text{init1}, 1, 7 \rangle \rangle \]

is instantiated to

\[ \langle \text{hasType}(\text{method}2), \langle \text{init1}, 1, 7 \rangle \rangle \]

because a1init is an interface initial value and it a possible concrete value of a1init (bound at call site 8).

The set of def-uses computed by Phase III for a library is a superset of the precise set of def-uses for the library; this is expected because computing the precise solution is undecidable.7

**Implications for testing:** The reason for retaining the relevant contexts in the final def-use solution is that for a library L, the set of relevant contexts of the def-use relationships computed during Phase III capture (up to the user specified bounds on the number of conjuncts of different kinds in any relevant context) all the conjunctions of potential aliases, potential non-aliases and type constraints involving interface initial values that are relevant for data-flow-based unit testing of L and need to be exercised by test cases. For example, the type constraint (\(\text{type}(a1init) \in A::\text{update} \)) is relevant for data-flow-based unit testing of the public interface method method2 of Lexample because it forms the relevant context of a def-use relationship (as shown in the previous paragraph) and hence needs to be exercised in a test case; however, the concrete type of a1init is not relevant for data-flow-based unit testing of method2 because no type constraint involving a1init is part of the relevant context of any def-use relationship. If only the def-use solution is required then the relevant contexts can be dropped from the final solution.

6 Algorithm for finding reduced context cover

In this section we discuss some ways in which the relevant contexts of def-use relationships computed during Phase III can be used in designing relevant test cases for satisfying testing criteria based on def-use relationships.

Let C be a set of conjunctions of potential aliases, potential non-aliases and type constraints. A set T is called a reduced context cover of C if and only if:

1. \(\text{size-of}(T) \leq \text{size-of}(C)\);
2. for each \(c \in C\), \(\exists t \in T \wedge t \Rightarrow c\); and
3. for each \(t \in T\), \(\exists c \in C \wedge t \Rightarrow c\).

Let RelevantContextCoverC be the set of relevant contexts of the def-use relationships computed by Phase III for a library L. Let RelevantContextCoverM be the subset of RelevantContextCoverC that involves the interface initial values of a public interface method M of L. Consider a call to M in a test case. A tester can completely specify the values of globals and parameters at the call site of M or alternatively, may choose to partially specify these values, and thus provide a test case template for generating many test cases. In the latter case, at the call to M the tester may choose to (1) indicate that v is completely unspecified or (2) specify the value of v using interface initial values, for v a global or parameter. Pointer fields can be specified in one of the following ways: (1) indicate that it is completely unspecified, (2) specify that it is null or (3) specify its value using another interface initial value. In contrast, the values of non-pointer variables need to be specified completely because Def-Use-Algo does not calculate their values. The tester may optionally choose a type for each interface initial value as well.

Given a template t for a call to M, let CoverM be the subset of RelevantContextCoverM that is consistent with the specification of t. One strategy for the tester is to manually enhance t using each element of CoverM, and instantiate each enhanced template into a test case. This will result in size-of(CoverM) test cases which can be executed separately. However, there is a more efficient alternative: first the tester can compute a reduced context cover ReducedCoverM of CoverM having as small a size as possible and then generate size-of(ReducedCoverM) test cases by enhancing t using the elements of ReducedCoverM. Intuitively, the goal is to cover as many def-use relationships as possible starting with the partial specification given by the tester. Note that the execution of some def-use paths require that non-pointer variables have specific initial values and hence their coverage cannot be guaranteed unless these values are assigned by the tester. A tester is free to instantiate each enhanced test template in any way as long as the specified constraints are satisfied.

Calculating an efficient cover: Given a template t of a call to a public interface method M and CoverM, the following simple greedy algorithm finds a reduced context cover ReducedCoverM such that each element of ReducedCoverM is consistent with t. The algorithm may not find the minimal cover in some cases.

\[
\text{ReducedCoverM} = \phi \\
\text{for each } s \in \text{CoverM} \\
\text{if } s \text{ contains a representative unknown initial value} \\
\text{ReducedCoverM} = \text{ReducedCoverM} \cup \{s\} \\
\text{else} \\
\text{if } (\exists r \in \text{ReducedCoverM} \text{ such that } r \wedge s \text{ can be satisfied and } r \wedge s \text{ is consistent with } t) \\
\text{\quad ReducedCoverM} = \text{ReducedCoverM} \cup \{r \wedge \{\text{new-e1}\}\} \\
\text{else} \\
\text{ReducedCoverM} = \text{ReducedCoverM} \cup \{s\}
\]

For example, RelevantContextCoverLexample for the library Lexample given in Figure 1 is

\{ (\text{empty}), (\langle a1init eq a5init \rangle), (\langle a1init neq a5init \rangle), (\langle \text{type}(a1init) \in A::\text{update} \rangle), (\langle \text{type}(a1init) \in C::\text{update} \rangle), (\langle \text{type}(e2init) \leq ET2 \rangle), (\langle \text{type}(e2init) \leq ET4 \rangle), (\langle \text{type}(e2init) \not\leq ET2 \rangle \wedge (\langle \text{type}(e2init) \not\leq ET4 \rangle) \}.

The relevant def-uses whose relevant contexts are the non-empty elements of RelevantContextCoverLexample, are shown in Figure 5. Figure 4 shows the def-uses computed by Phase I and Figure 5 shows the same def-uses after the unknown initial values and unknown initial defs have been instantiated with concrete values in Phase III.

Now consider a template t1 of a call to public interface method method2 of Lexample where all pointer variables are left unspecified by the tester. RelevantContextCoverMmethod2 is \{ (\text{empty}), (\langle a1init eq a5init \rangle), (\langle a1init neq a5init \rangle), (\langle \text{type}(a1init) \in A::\text{update} \rangle), (\langle \text{type}(a1init) \in C::\text{update} \rangle) \} and

\[
7\text{It is possible to automate this step to a great extent and we plan to investigate this in future.}
\]
\textit{Cover}_{method2} is the same as \textit{RelevantContextCover}_{method2}. The reduced context cover \textit{ReducedCover}_{method2} computed by the algorithm given above is
\[
\{(a_{init} \text{eq} a_{init}) \land (type(a_{init}) \in A::update)), \\
(a_{init} \text{neq} a_{init}) \land (type(a_{init}) \in C::update))\}.
\]
Consider the test case that can be generated from \(t_1\) after enhancing \(t_1\) with the first condition of the reduced cover.

```java
public void test1() {
    A var = new A();
    method2(var, var, null);
}
```

Similarly, the following test case can be generated from \(t_1\) after enhancing \(t_1\) with the second condition.

```java
public void test2() {
    C var1 = new C();
    A var2 = new A();
    method2(var1, var2, null);
}
```

7 Discussion

Finally: In Java each try statement can optionally have a finally statement associated with it; this must be executed [GJS96] no matter how the try statement terminates: with an exception or without an exception. As stated earlier, for the ease of presentation we have ignored finally statements in this paper. These can be easily accommodated using the approach described in [Cha99]. Intuitively an additional kind of \(\text{ECFInfo}\) is needed to capture control flow when a try statement terminates normally or when it terminates due to a labelled break or continue. A call or a try statement nested inside a finally can cause exceptions to stack up; however, singleton \(\text{ECFInfo}\) is still enough because the stack is implicitly maintained at call sites by storing \(\text{ECFInfo}\) in each actual-to-unknown-initial-value binding, and a try nested inside a finally is treated like a call to an anonymous procedure.

Run-time Exceptions: We have considered only exceptions generated by \textit{throw} statements in this paper; since run-time exceptions can be generated by almost any statement, we have ignored them. Our algorithm can handle run-time exceptions if the set of statements that can generate these exceptions is given as an input. If all statements that can potentially generate run-time exceptions are considered, we will get a safe solution, however this may generate far more information than what is useful.

Unknown procedures: In this paper we have assumed that all the procedures invoked by a library are available. Note that we can analyze a library without the availability of any driver for the library, which is typically the case during unit testing of a library. If a library invokes an unknown procedure external to the library, then a user generated stub for the procedure can be used.

Program Understanding: The alias contexts, type contexts and exception context’s computed by Def-Use-Algo can also be used by program understanding tools. Def-Use-Algo can help by uncovering unexpected but relevant potential aliases, potential non-aliases and type constraints. Moreover, the complexity of alias contexts, type contexts and exception context’s provide a measure of code complexity. Complicated alias contexts, type contexts or exception context’s point out portions of code that are hard to maintain and understand. The investigation of these issues is part of future work.

Implementation: We have implemented a prototype of \textit{Relevant Context Inference} for points-to analysis of C++ programs. The results, presented in [CRL99], are encouraging and argue for the effectiveness of this technique in practice. The implementation of the rest of Def-Use-Algo is part of future work. We also plan to incorporate Def-Use-Algo in a tool for generating test cases and measuring def-use coverage of O-O libraries. The library needs to be instrumented to observe def-use relationships executed by a set of test cases.

8 Related Work

Data-flow-based testing [DGS96, FO76, LK83, FW93, HL91, Wey94] has been advocated since 1985 to supplement control-flow based methods. Coverage metrics for def-uses have been discussed [RW85, FW88, OW91, LCS89] and coverage checking tools exist [PFW85, Ost90].

Def-use analyses for C [PLR94] and Fortran [HS94] have been developed. Recent work explores def-use testing in object-oriented programs [HR94, SP00]. Rothermel and Harrold describe a data-flow-based testing technique based on intra-method and inter-method def-uses within a C++ class. Souter and Pollock describe a testing strategy which uses an extended version of Whalley and Rinard’s points-to/escape graph [WR99] to track reads and writes into object fields. Neither of these algorithms handle flow due to exceptions.

Sinha and Harrold [SH99] discuss unit and integration testing of classes containing explicitly thrown exceptions. Incomplete programs are tested using stubs and drivers. Exception coverage criteria are defined that exercise def-uses of exception objects. In contrast to our use of exception contexts, their approach is to record exceptional control flow explicitly in the program representation.

Recent work in our research group presents a theoretical model for analysis of partial programs [RRL99]. Experiments with separate points-to and side effect analyses of C libraries and library clients are described in [RR01].

Recall that Def-Use-Algo is an application of relevant context inference to def-use analysis. The points-to analysis phases of Def-Use-Algo are similar to those in [CRL99], but are extended to include exception contexts, which were ignored in our prototype. Thus, the main contributions of Def-Use-Algo over our previous work include: design of def-use analysis, handling of incomplete object-oriented programs (i.e., libraries) and the handling of explicitly thrown exceptions.

9 Conclusions

Data-flow-based testing of object-oriented libraries is difficult because of unknown aliasing between parameters, unknown concrete types of the parameters, dynamic dispatch and exceptions. We have presented the first algorithm for finding def-uses in libraries written in C++/Java that overcomes the above difficulties. We have also shown that relevant test cases for unit testing of object-oriented libraries can be generated by enhancing user generated test-case templates using the information computed by our algorithm.

Acknowledgements We thank Phyllis Frankl for providing pointers to relevant literature, and we thank Tom Marlowe and Dave Wonnacott for reviewing an earlier draft of this paper. The research reported here was supported, in part, by NSF grant CCR-9501761 and the Hewlett-Packard Corporation.
6:1 \[(a_{\text{init}} \text{ init} a_2_{\text{init}}), \text{empty}, \text{empty}, (a_{\text{init}}\_next, 4, 6)\]
6:2 \[(a_{\text{init}} \text{ neq} a_2_{\text{init}}), \text{empty}, \text{empty}, (a_{\text{init}}\_next, 5, 5, 6)\]
7:1 \[
\{\text{empty}, (\text{type}(a_{\text{init}}) \in A)\_update, \text{empty}, (\text{global}1, 1, 7)\}
\]
7:2 \[
\{\text{empty}, (\text{type}(a_{\text{init}}) \in C)\_update, \text{empty}, (\text{global}1, 2, 7)\}
\]
17:1 \[
\{\text{empty}, \text{empty}, (\text{type}(e_{\text{init}}) \leq ET2), (\text{global}3, 13, 17)\}
\]
25:1 \[
\{\text{empty}, \text{empty}, (\text{type}(e_{\text{init}}) \leq ET4), (\text{global}3, 13, 25)\}
\]
27:1 \[
\{\text{empty}, \text{empty}, (\text{type}(e_{\text{init}}) \notin ET2) \land (\text{type}(e_{\text{init}}) \notin ET4), (\text{global}3, 13, 27)\}
\]

Figure 4: Part of Phase I def-use solution (dfelms-e)

/**/ \text{interface initial values } a_{\text{init}} \text{ and } a_5_{\text{init}} \text{ are respectively concrete values of } a_{\text{init}} \text{ and } a_2_{\text{init}} \text{ at call site } 8, \text{ and initial def } a_5_{\text{init}}\_next\_initdef \text{ is the concrete value of } a_2_{\text{init}}\_next\_initdef \text{ at call site } 8 \text{ */}/
6:1 \[(a_{\text{init}} \text{ eq} a_5_{\text{init}}), \text{empty}, \text{empty}, (a_5_{\text{init}}\_next, 4, 6)\]
6:2 \[(a_{\text{init}} \text{ neq} a_5_{\text{init}}), \text{empty}, \text{empty}, (a_5_{\text{init}}\_next, 5, 5, 6)\]
7:1 \[
\{\text{empty}, (\text{type}(a_{\text{init}}) \in A)\_update, \text{empty}, (\text{global}1, 1, 7)\}
\]
7:2 \[
\{\text{empty}, (\text{type}(a_{\text{init}}) \in C)\_update, \text{empty}, (\text{global}1, 2, 7)\}
\]
17:1 \[
\{\text{empty}, \text{empty}, (\text{type}(e_{\text{init}}) \leq ET2), (\text{global}3, 13, 17)\}
\]
25:1 \[
\{\text{empty}, \text{empty}, (\text{type}(e_{\text{init}}) \leq ET4), (\text{global}3, 13, 25)\}
\]
27:1 \[
\{\text{empty}, \text{empty}, (\text{type}(e_{\text{init}}) \notin ET2) \land (\text{type}(e_{\text{init}}) \notin ET4), (\text{global}3, 13, 27)\}
\]

Figure 5: Part of Phase III def-use solution

References

for each SCC in reverse topological order
mark-reachable()
phase-I-pta()  
*** following three functions perform Phase I-dua ***
mark-used-fields()
initialize-worklist-I-dua()
process-worklist-I-dua()

Figure 6: Phase I

A Propagation of Data-flow Elements

This appendix presents pseudo-code for Phases I and II. The different steps of Def-Use-Algo are illustrated using the example in Figure 1.

A.1 Phase I

This section presents the def-use-analysis transfer function for each kind of ICFG node. We A high-level pseudocode for Phase I is given in Figure 6. For example each SCC contains exactly one method and \{A::update, C::update, method1, method2, method3, method4\} is a reverse topological ordering of the SCC’s.

mark-reachable[CRL98] marks reachable ICFG nodes in the current SCC using a simple algorithm. It assumes the entry node of each method in the current SCC is reachable. We skip its details here as these are not essential to this paper. For the example in Figure 1, all nodes are marked reachable by mark-reachable.

phase-I-pta is same as Phase I in [CRL99, CRL98]. For this paper it is enough to know that phase-I-pta iteratively computes, in terms of PTA-defelms, a points-to solution at each node in the current SCC. We will present where relevant the solution computed by Phase-I-pta for the example in Figure 1.

mark-used-fields, initialize-worklist-I-dua and process-worklist-I-dua perform Phase I-dua.

mark-used-fields iteratively analyzes the current SCC to mark all the non-pointer fields of unknown initial values that are defined or read. This is needed because non-pointer fields are not considered in Phase I-pta. Iteration is needed because due to actual-to-actual bindings, the use of a field in a callee may imply the use of another field in the caller. Note that Phase I-pta has already marked the pointer fields of unknown initial values that have been found to be used. For example, in method1 in Figure 1, Phase I-pta marks a1\_init\_next as defined and a2\_init\_next as read. Since a3\_init\_next is not used in method1 or in any method invoked during the lifetime of method1, a3\_init\_next is not marked.

initialize-worklist-I-dua (Figure 7) initializes the worklist with the definitions generated at entry nodes, and reachable (marked by mark-reachable) assignment nodes, throw nodes, object creation sites and call nodes. The formal definitions of the functions invoked by initialize-worklist-I-dua are given in Appendix B; here we will explain them informally.

propagate-unknown-init-defs-from-entry-node (Figure 7) initializes the worklist with dfelms-c representing defs of parameters, and unknown initial def of globals and fields of unknown initial values; it considers only those parameters, locals and fields of unknown initial values that have been found to be read.

propagate-defs-from-assignment-node (Figure 7) initializes the worklist with dfelms-c representing defs generated at reachable assignment nodes, by using the points-
figure 7: initialize worklist for phase i-dua

to solution computed by Phase-I-dua. For each def generated by an assignment node, it also computes the potential defs due to potential aliases at the entry of a method. For example, at statement 4 in Figure 1, using the points-to solution it generates the def \{empty,Empty-context,\{a1init.next,4,\{Empty-context,a3init\}\}\}, and this def implies the potential def \{empty,\{a1init eq a2init,empty,empty,\{a2init.next,4,\{Empty-context,a3init\}\}\\}, both of which are added to the worklist.

**propagate-exception-objects-from-throw-node** (Figure 7) initiates the propagation of dfelms-c representing definitions of exception objects. It uses the points-to solution to determine the exception objects thrown by n. For example, at statement 14 in Figure 1, there is only one possible exception object, e1init; hence, \{Empty-context, e1init,4\} is propagated to the exit node of innermost-enclosing-exception-block(4), i.e., program point 15, the exit node of the try statement.

**propagate-defs-created-at-an-object-creation-site** (Figure 7) initializes the worklist with a dfelm-c representing the assignment of the newly created object to the left-hand-side variable. It also initializes with null the fields of the heap-name representing the objects allocated at the object creation site, and adds dfelm-c representing these definitions to the worklist.

**back-bind-using-def-use-summary-transfer-function** (Figure 7) instantiates the DUA-dfelm at the exit node of a non-same-SCC method m using actual-to-uv binds at a call site of m in the current SCC, and propagates the instantiations to the successor of the call site. It uses the Phase I-pta points-to solution to determine the methods invocable from a dynamically dispatched call site. Actual-to-uv binds are computed by phase-I-pta. For example, [empty,Empty-context,\{a1init.next,4,\{Empty-context,a3init\}\}] reaches the exit of method1 and hence this dfelm-c is part of the def-use summary transfer function of method1. Thus, when Phase I-dua is done on method2, at call site 8, the above dfelm-c is instantiated to [empty,Empty-context,\{a4init.next,4,\{Empty-context,a6init\}\}].

**process-worklist-I-dua** (Figure 8) invokes the def-use-analysis node transfer functions discriminating by node type.

**Def-use-analysis transfer functions** 1, 2, 3. Assignment, condition and object creation nodes (Figure 9): Figure 9 defines the def-use-analysis transfer functions of assignment, condition and new (object creation site) nodes. generate-dfelm-due-to-pot-non-aliases(n, rdfe) (Appendix C) conditions the propagation of rdfe across n on potential non-aliases. For example, when rdfe is the dfelm-c [empty,Empty-context,\{a2init.next,a2init.nextinsidef,\{Empty-context,a2init.nextinit\}\}] at statement 4 in Figure 1, generate-dfelm-due-to-pot-non-aliases generates the set of dfelm-c \{\{empty,\{a1init neq a2init,empty,empty,\{a2init.next,a2init.nextinsidef,\{Empty-context,a2init.nextinit\}\}\}\, \}

4. process-call (Figure 9): 4.a. compute-p-uses

due-to-dynamic-dispatch uses val to compute the p-uses due to dynamic dispatch mentioned in Section 1. For example, when rdfe is [empty,Empty-context,\{r,3,\{Empty-context,a1init\}\}] at statement 5 in Figure 1, val is a1init and the possible targets with a1init as receiver are A::update and C::update. As a result, the dfelm-g [[empty,\{type(a1init)\}]A::update,empty,\{r,3,\{A::update\}\}] and [[empty,\{type(a1init)\}]C::update,empty,\{r,3,\{5,C::update\}\}] are added to 5.def-uses.
1. process-assignment( n, rdfe )
   if ( var is read in n )
   n.def-uses = n.def-uses \cup \{ rc1,\{ var,def,(n.true-successor) \} \}
   n.def-uses = n.def-uses \cup \{ rc1,\{ var,def,(n.false-successor) \} \}
   for each successor succ of n
   add-to-soln-and-worklist-if-needed-I( succ,\{ rdfe \} )

2. process-condition( n, rdfe )
   if ( var is read in n )
   add-to-soln-and-worklist-if-needed-I( node, dfe )
   \begin{itemize}
   \item \textbf{a:} compute-p-uses-due-to-dynamic-dispatch( n, rdfe )
   \item \textbf{b:} compute-new-bindings-of-defs-and-u-i-defs( n, rdfe )
   \item \textbf{c:} compute-p-uses-due-to-dynamic-dispatch( n, rdfe )
   \item \textbf{d:} compute-new-bindings-of-defs-and-u-i-defs( n, rdfe )
   \item \textbf{e:} compute-p-uses-due-to-dynamic-dispatch( n, rdfe )
   \item \textbf{f:} compute-new-bindings-of-defs-and-u-i-defs( n, rdfe )
   \end{itemize}

3. process-new( n, rdfe )
   if ( var is read in n )
   n.def-uses = n.def-uses \cup \{ rc1,\{ var,def,(n.true-successor) \} \}
   n.def-uses = n.def-uses \cup \{ rc1,\{ var,def,(n.false-successor) \} \}
   for each successor succ of n
   add-to-soln-and-worklist-if-needed-I( succ,\{ rdfe \} )

4. process-call( n, rdfe )
   if ( var is an actual )
   n.def-uses = n.def-uses \cup \{ rc1,\{ var,def,(n.true-successor) \} \}
   n.def-uses = n.def-uses \cup \{ rc1,\{ var,def,(n.false-successor) \} \}
   \begin{itemize}
   \item \textbf{a:} compute-p-uses-due-to-dynamic-dispatch( n, rdfe )
   \item \textbf{b:} compute-new-bindings-of-defs-and-u-i-defs( n, rdfe )
   \item \textbf{c:} compute-p-uses-due-to-dynamic-dispatch( n, rdfe )
   \item \textbf{d:} compute-new-bindings-of-defs-and-u-i-defs( n, rdfe )
   \item \textbf{e:} compute-p-uses-due-to-dynamic-dispatch( n, rdfe )
   \item \textbf{f:} compute-new-bindings-of-defs-and-u-i-defs( n, rdfe )
   \end{itemize}

5. process-try-exit (Figure 10)
   For each method invocation on n throws an exception that is not caught within the target, a new dfelm-c is computed whose ECFInfo stores the signature of the exception thrown by the target. If expr-object is a heap-name, get-ecfi(expr-object) returns type(expr-object). Otherwise expr-object is an unknown initial value and get-ecfi(expr-object) returns expr-object. The exceptions thrown by targets of n and the relevant contexts under which these exceptions are thrown are determined using the Phase I-pta solution.

6. process-throw (Figure 10)
   For each method invocation on n throws an exception that is not caught within the target, a new dfelm-c is computed whose ECFInfo stores the signature of the exception thrown by the target. If expr-object is a heap-name, get-ecfi(expr-object) returns type(expr-object). Otherwise expr-object is an unknown initial value and get-ecfi(expr-object) returns expr-object. The exceptions thrown by targets of n and the relevant contexts under which these exceptions are thrown are determined using the Phase I-pta solution.

Figure 9: Transfer functions for assignment, condition, new and call nodes

4.b. compute-new-bindings-of-defs-and-u-i-defs computes the bindings between the unknown initial defs at the entry nodes of the potential targets of n and the values of these unknown initial defs at n. Each def-uidef binding at n has the form \( rc_1,\{ var,def,(rc_2,val) \} \). Here rc is a relevant context under which the binding holds, uidef is an unknown initial def at the entry of a target of n, var is a location whose reaching definitions are represented by uidef at the entry of the target, and def is a definition point or an unknown initial def that represents a definition of var. The reason why var is also stored in the binding is to increase precision in those cases where the reaching definition is that of a field of a heap-name or an unknown initial value. This ensures that the different objects and defs reaching the entry node of a method along different paths are not mixed. This is a technical detail whose usefulness will be become clear later in Sections A.2 and 5. n.def-uidef-bindings contains these bindings between the unknown initial defs and their values at n. For example in Figure 1, \[ \{ Empty-context,\{ object_10,\{ next,10,\{ a_2init,\{ next_init_1 \} \} \} \} \in n.def-uidef-bindings. \]

Details of compute-p-uses-due-to-dynamic-dispatch and compute-new-bindings-of-defs-and-u-i-defs are given in Appendix D.

If any of the targets invocable from n throws an exception that is not caught within the target, a new dfelm-c is computed whose ECFInfo stores the signature of the exception thrown by the target. If expr-object is a heap-name, get-ecfi(expr-object) returns type(expr-object). Otherwise expr-object is an unknown initial value and get-ecfi(expr-object) returns expr-object. The exceptions thrown by targets of n and the relevant contexts under which these exceptions are thrown are determined using the Phase I-pta solution.

A.2. The reason why var is also stored in the binding is to increase precision in those cases where the reaching definition is that of a field of a heap-name or an unknown initial value. This ensures that the different objects and defs reaching the entry node of a method along different paths are not mixed. This is a technical detail whose usefulness will be become clear later in Sections A.2 and 5. n.def-uidef-bindings contains these bindings between the unknown initial defs and their values at n. For example in Figure 1, \[ \{ Empty-context,\{ object_10,\{ next,10,\{ a_2init,\{ next_init_1 \} \} \} \} \in n.def-uidef-bindings. \]

Details of compute-p-uses-due-to-dynamic-dispatch and compute-new-bindings-of-defs-and-u-i-defs are given in Appendix D.

If any of the targets invocable from n throws an exception that is not caught within the target, a new dfelm-c is computed whose ECFInfo stores the signature of the exception thrown by the target. If expr-object is a heap-name, get-ecfi(expr-object) returns type(expr-object). Otherwise expr-object is an unknown initial value and get-ecfi(expr-object) returns expr-object. The exceptions thrown by targets of n and the relevant contexts under which these exceptions are thrown are determined using the Phase I-pta solution.
15 in Figure 1, the exit node of the try statement in method3. Since type of e1_init is compatible with ET2, successors is (16) and rdfe is propagated to the entry node of the catch at program point 16. Moreover, the exception (e1_init) can escape the catch at statement 16 because the concrete type of the exception can be ET1 or ET4. As a result, get-escape-tcs returns (type(e1_init) = ET2) and the dfelm-c [e1_init, empty, empty, (type(e1_init) = ET2)], (global3, 13, {Empty-context, object13}) is propagated to program point 18, i.e., exit node of innermost-enclosing-exception-block(15). Next, (second case) let rdfe be the dfelm-d [Empty-context, e1_init, 14] at program point 15. Due to the same reasons as in the first case, rdfe is propagated to program point 16 and the dfelm-d [{empty, empty, (type(e1_init) = ET2)], e1_init, 14} is propagated to program point 18.

7. process-catch (Figure 11): If rdfe represents a variable definition, process-catch catches the exception represented by the ECFInfo of rdfe. If rdfe represents an exception object, process-catch generates a def-use relationship between the throw statement that threw the exception object and the entry node of the catch statement. In the second case, process-catch also generates a dfelm-c representing a definition of the parameter of the catch statement because the parameter is assigned the caught exception object. If ECFInfo or expc-object is an unknown initial value, get-catch-tcs is used to generate the appropriate type constraints under which the exception is caught by the catch statement. For example, consider program point 16 in Figure 1. Consider two different rdfe’s at this program point: dfelm-c [e1_init, Empty-context, (global3, 13, {Empty-context, object13})] and dfelm-d [Empty-context, e1_init, 14]. In the first case, ECFInfo is e1_init and hence get-catch-tcs returns (type(e1_init) ≤ ET2). As a result, new-dfe is the dfelm-c [empty, empty, empty, (type(e1_init) = ET2)], (global3, 13, {Empty-context, object13}). In the second case, expc-object is e1_init and again get-catch-tcs returns (type(e1_init) ≤ ET2). As a result, the dfelm-f [{empty, empty, empty, (type(e1_init) ≤ ET2)], e1_init, 14, 16} is added to the solution at program point 16 and new-dfe = empty, empty, empty, (type(e1_init) ≤ ET2)), (excp, 16, {Empty-context, e1_init}) is generated to represent a definition of the parameter of the catch statement.

8. process-catch-exit (Figure 11): If rdfe flows to n along a path without any uncaught exception (i.e., ECFInfo is empty), process-catch-exit propagates rdfe to the successor of the try-catch construct. Otherwise, process-catch-exit propagates rdfe to the exit node of innermost-enclosing-exception-block(n).

9.10. Return sites and method exit nodes (Figure 13): Figure 13 in Appendix E defines def-use-analysis transfer functions for return sites (i.e., successors of call nodes) and method exit nodes.

11. Return statements (Figure 14): Figure 14 in Appendix E defines def-use-analysis transfer function for return statements.

A.2 Phase II

A high-level description of Phase II is given in Figure 12. phase-II-pta is same as the Phase II described in [CRL99, CRL98]. For this paper it is enough to know that phase-II-pta iteratively propagates concrete values of unknown initial values to the entry nodes of methods. In order to avoid propagation of concrete values from unreachable methods, Def-Use-Algo computes an initial set of reachable methods,

Figure 10: Transfer functions for throw and try-exit
7.a. CATCH-VARIABLE-DEF(n, rdfe) {
    if (rdfe is [exp-type,rc1,(var,def,(rc2,va2))])
        new-dfe = [empty,rc1,(var,def,(rc2,va2))]
    else
        new-dfe = [empty,rc1,(var,def,(rc2,va2))]
    return new-dfe
}

7.b. CATCH-EXCEPTION-OBJECT(n, rdfe) {
    if (exp-obj is a heap-name)
        new-rc = rc1
    else
        new-rc = rc1
    new-dfe = [empty,new-rc,(var,def,(rc2,va2))]
    return new-dfe
}

8. process-catch-exit(n, rdfe) {
    if (first case and var is the parameter of the catch statement)
        successor = exit node of innermost-enclosing-exception-block(n)
        add-to-soln-and-worklist-if-needed-I(successor,{rdfe})
    else
        if (first case and exp-obj is empty)
            successor = ordinary successor of n
        else
            successor = exit node of innermost-enclosing-exception-block(n)
            add-to-soln-and-worklist-if-needed-I(successor,{rdfe})
    }

Figure 11: Transfer functions for catch entry and catch exit nodes
for each SCC in topological order perform
phase-II-pta();
// following two functions perform Phase II-dua
1: initialize-worklist-II-dua();
2: process-worklist-II-dua();

1. initialize-worklist-II-dua() {
    worklist = empty
    for each reachable method \( m \) in the current SCC {
        for each binding \( \in m\).reaching-defs
            \( \text{wl-node} = \text{new worklist node} (m, \text{binding}) \)
        add \( \text{wl-node} \) to worklist
        for each reachable call node \( c \) in \( m \) {
            for each def-uidef binding
                \( [r_c,(\text{var},\text{def}),\text{uidef}] \in c\).def-uidef-bindings \{
                    if \( \text{def} \) is a program point or an interface initial def
                        \( \text{result} = \text{instantiate-def-uidef-binding}(m, [r_c,(\text{var},\text{def}),\text{uidef}]) \)
                    // uidef1.method is the target (of \( c \)) with
                    // whose entry node uidef1 is associated
                    add-to-soln-and-worklist-if-needed-II(uidef1.method, result)
                } 
            } 
        } 
    } 
}

1.a instantiate-def-uidef-binding( method, binding ) {
    \( \text{result} = \emptyset \)
    let \( \text{binding} = [r_c,(\text{var},\text{def}),\text{uidef}] \)
    for each instantiation of UIV’s in \( (r_c,\text{var}) \) with their concrete values computed at
    the entry node of \( \text{method} \) during Phase II-pta { 
        if \( r_c \) can be true after the instantiation
            let \( \text{var1} \) be the value of \( \text{var} \) after the instantiation
            let \( \text{rc1} \) be the value of \( r_c \) after the instantiation
            \( \text{result} = \text{result} \cup \{[rc1,(\text{var1},\text{def}),\text{uidef}]\} \)
    } 
    return \( \text{result} \)
}

2. process-worklist-II-dua() {
    while worklist is not empty {
        \( \text{wl-node} = \text{delete from work-list} \) //*** \( \text{wl-node} = (\text{method, binding}) ***/ 
        \( m = \text{wl-node.method} \)
        \( \text{binding} = \text{wl-node.binding} \)
        let \( \text{binding} = [r_c1,(s,\text{def}),\text{uidef}] \)
        for each reachable call node \( c \) in \( m \) {
            for each def-uidef binding
                \( [r_c2,(\text{var},\text{uidef}),\text{uidef}2] \in \text{c.def-uidef-bindings} \{
                    \( \text{rc3} = \text{rc1} \land \text{rc2} \)
                    \( \text{new-rcs} = \)
                    instantiate-rc-under-constraint(\( m, \text{rc3}, \text{var}, \text{s} \))
                    for each \( r_c \in \text{new-rcs} \)
                        reaching-def = \( [r_c,(\text{var},\text{def}),\text{uidef}2] \)
                        add-to-soln-and-worklist-if-needed-II(uidef2.method, \{reaching-def\})
                } 
        } 
        add-to-soln-and-worklist-if-needed-II( \( \text{method}, \text{bindings} \) ) {
            for each binding \( \in \text{bindings} \)
                if binding \( \notin \text{method.reaching-defs} \)
                    method.reaching-defs = method.reaching-defs \cup \{binding\}
                if (\( \text{method is in the current SCC} \) )
                    \( \text{wl-node} = \text{new worklist node} (\text{method, binding}) \)
                    add \( \text{wl-node} \) to worklist
        }
    }
}

Figure 12: phase II
B Auxiliary functions of init-worklist-I-dua

1. propagate-unknown-init-defs-from-entry-node(n) {
   /**** it propagates dfelms-c-***/  
   for each field u.f  where u is an UV at n and u.f  
   has been marked as read  
   y={empty,Empty-context,(u.f,init-dua,(Empty-context,u.f))}  
   add-to-soln-and-worklist-if-needed-I(n.successor, (y))  
   /**** n is the entry node of n.method ****/  
   for each global g read in n.method or any method  
   invoked from n.method directly or indirectly  
   during its lifetime  
   y = {empty,Empty-context,(g,Binit-dua,(Empty-context,ginit))}  
   add-to-soln-and-worklist-if-needed-I(n.successor, (y))  
   for each parameter p of n.method read in n.method  
   y = {empty,Empty-context,(p,Empty-context,pinit))}  
   add-to-soln-and-worklist-if-needed-I(n.successor, (y))  
}

2. propagate-defs-from-assignment-node(n) {  
   gen-dfelm-c = φ  
   for each (rc1, var) in rhs-rc-loc-pairs  
   for each (rc2, val) in rhs-rc-loc-pairs  
   gen-dfelm-c = gen-dfelm-c ∪ {y}  
   new-dfe = [eqf,rc3,⟨u.f,def,(rc2,rc1)⟩]  
   generated-dfes = generated-dfes ∪ {new-dfe}  
}  

return generated-dfes }

3. propagate-exception-objects-from-throw-node(n) {  
   for each exception object eo thrown by n  
   let rc3 be the relevant context under which eo  
   is thrown by n  
   gen-dfelm-c = gen-dfelm-c ∪ {rc3, eo,n}  
   successor = exit node of innermost-enclosing-exception-block(n)  
   add-to-soln-and-worklist-if-needed-I(n.successor, gen-dfelm-c)  
}

4. propagate-defs-created-at-an-object-creation-site(n) {  
   gen-dfelm-c = φ  
   y = {empty,Empty-context,(n, lbs.n, (Empty-context,object-n))}  
   gen-dfelm-c = gen-dfelm-c ∪ {y}  
   for each field f of object-n  
   y = {empty,Empty-context,(object-n,f,Empty-context,null)}  
   gen-dfelm-c = gen-dfelm-c ∪ {y}  
   add-to-soln-and-worklist-if-needed-I(n.successor, gen-dfelm-c)  
}

5. back-bind(using-def-use-summary-transfer-function(m, c) {  
   for each dfe ∈ m.exit-node.reaching-defs  
   /**** dfe is dfelm-c [eqf,rc1,⟨var,def,(rc2,val)⟩] or  
   dfe is dfelm-c [rc1,excp-obj,throw-point] ****/  
   if ((dfe is a dfelm-d) or  
   (var is not a local variable and  
   def is a program point))  
   x = back-bind(dfe, c, m, exit-node)  
   /**** back-bind returns  
   [eqf,rc3,⟨var,def,(rc2,val)⟩]’s  
   or [rc3,excp-obj,throw-point]’s  
   to which dfe maps at c ****/  
   z = x  
   for each dfelm-c dfe ∈ x  
   z = z ∪  
   generate-defs-due-to-pot-aliases(c.method,dfe)  
   add-to-soln-and-worklist-if-needed-I(c.successor, z)  
}

propagate-defs-from-assignment-node initializes worklist with dfelms-c representing defs generated at reachable assignment nodes. lhs-rc-loc-pairs contains pairs of the form  
(rc, loc), where loc is a location represented by left-hand-side  
and rc is the relevant context under which lhs represents this  
location. If lhs is a variable name, say y, then lhs-rc-loc-pairs  
is { ⟨Empty-context,p⟩ }. The meaning of rhs-rc-loc-pairs is  
similar except that it represents values of right-hand-side. If  
right-hand-side is of arithmetic type, i.e., it does not repre-  
sent any address, then rhs-rc-loc-pairs is { ⟨Empty-context,  
don’t-care⟩ }. lhs-rc-loc-pairs and rhs-rc-loc-pairs are computed  
using the points-to solution computed by phase-I-pta. For example, at statement 4 in Figure 1, lhs-rc-loc-pairs is  
{ ⟨Empty-context,a1init.next⟩ } and rhs-rc-loc-pairs is  
{ ⟨Empty-context,a3init⟩ }. Similarly, at statement 6 in  
Figure 1, lhs-rc-loc-pairs is  
{ ⟨Empty-context,global2⟩ } and rhs-rc-loc-pairs is  
{ ⟨(a1init eq a2init),empty,empty,a3init⟩,  
((a1init neq a2init),empty,empty,a2init.next,a3init) }  
This is because statement 4 modifies a2init.next if and only if  
a1init and a2init are the same object at the entry of method1,  
and it does not modify a2init.next if and only if a1init and  
a2init are not the same object at the entry of method1. As  
a result, Phase 1-pta computes the dfelms-a  
{empty,⟨(a1init eq a2init),empty,empty⟩,⟨a2init.next,a3init⟩}  
and  
{empty,⟨(a1init neq a2init),empty,empty⟩,  
(a2init.next,a3init.next,a3init.next) } on the top of statement 6.  
In Figure 1, generate-defs-due-to-pot-aliases generates  
dfelm-c representing definitions of fields of unknown initial  
values due to potential aliasing between compatible unknown  
initial values at the entry node of a method. For example,  
in Figure 1, generate-defs-due-to-pot-aliases generates  
dfelm-c [empty, ⟨(a1init eq a2init),empty,empty⟩,  
(a2init.next,⟨(empty,empty)⟩,Empty-context,0,a3init) ]  
when method is method2 and dfe is  
{empty,Empty-context,(a1init.next,⟨(empty,empty)⟩,Empty-context,0,a3init) }.  
back-bind(dfe, c, n) uses the bindings in c.actual-to-  
uv-bindings to instantiate the unknown initial values in dfe  
with their values at c. back-bind returns the set of DUA-  
dfelm resulting from the instantiations.
C generate-dfels-due-to-pot-non-aliases

1.a generate-dfels-due-to-pot-non-aliases(n, rdfe) {
   /***
    * rdfe is dfelm-a [rcf1,rc11,φ]  
    * n is an assignment node ***/
    * generated-dfels = φ
    * if ( n.lhs does not have dereference )
    * return ( rdfe )
    * Let n.lhs be p-f, where the declared type of p is A
    * if ( var is u where u is an UIV and the declared type of u is compatible with A )
    * for each (rc3, loc) in n.lhs-rec-loc-pairs
    * if ( loc is v.f and v is an UIV compatible with u )
    * rc4 = rc1 ∧ rc3 ∧ (loc eq u, empty, empty)
    * new-dfe = [rcf1,rc4,φ,def,rc2,val]
    * generated-dfels = generated-dfels ∪ (new-dfe)
    * else
    * generated-dfels = generated-dfels ∪ (rdfe)
    * else
    * generated-dfels = generated-dfels ∪ (rdfe)
    * return generated-dfels
   */
}

D Auxiliary functions for process-call

1.b compute-pu-defs-due-to-dynamic-dispatch(n, rdfe) {
   ```
   /***
    * rdfe is DUAP-elm-loc [rcf1,rc11,φ]  
    * n is a receiver variable ***/
    * if ( n does not represent a dynamic dispatch )
    * return
    * if ( val is an unknown initial value )
    * for each possible target t with val as receiver
    * cond = type constraint on type(val) under which
    * t is invoked with val as receiver
    * new-rec = rc1 ∧ rc2 ∧ (empty, empty, empty)
    * new-dfe = [rcf1,rc4,φ,def,rc2,val]
    * generated-dfels = generated-dfels ∪ (new-dfe)
    * else
    * generated-dfels = generated-dfels ∪ (rdfe)
   */
   ```
   let t be the target invoked with val as receiver
   new-rec = rc1 ∧ rc2
   n.def-uses = n.def-uses ∪ {new-rec,φ,def,rc2,val}
}

4.b compute-new-bindings-of-defs-and-u-i-defs(n, rdfe) {
   new-def-ui-def-bindings = φ
   if ( var is a global )
   4.b.1: uidefs = unknown initial def of var at the entry
   for each y ∈ uidefs
   if ( [rc1,φ,def,rc2,val] ∈ n.actual-to-ui-bindings
   uidefs = uidefs ∪ {rcf1,φ,def,rc2,val}
   })
   if ( var is object )
   4.b.2: uidefs = φ
   for each (rc1,φ,def,rc2,val) ∈ n.actual-to-ui-bindings
   uidefs = uidefs ∪ {rcf1,φ,def,rc2,val}
   })
   for each (rc3,loc,uidf) ∈ uidefs
   if ( uidefs has been marked as read )
   uidefs = uidefs ∪ {rcf3,loc,uidf}
   for each (rc4,φ,def,rc2,uidf) ∈ uidefs
   new-def-ui-def-bindings = new-def-ui-def-bindings ∪ {rcf4,loc,uidf}
   return new-def-ui-def-bindings
}

4.b compute-new-bindings-of-defs-and-u-i-defs (n, rdfe) :
For example, when rdfe is
{empty,Empty-context, (global1.method1.global1initdef,Empty-context.global1initdef)}
at statement 5 in Figure 1, uidefs (at program point 4.b.1)
is {A::update.global1initdef,C::update.global1initdef}, i.e., uidefs contains the unknown initial def of global1 at the entry nodes of A::update and C::update. Phase I-pta stores in n.actual-to-ui-bindings the bindings between the unknown initial values at the entry nodes of targets of n and their values at n; with each binding the relevant context under which the binding holds is also stored. For example, at program point 5 in Figure 1, 5.actual-to-ui-bindings is
{empty,(type(a1init)∈A::update),empty),global1initdef,A::update.global1initdef},
{empty,(type(a1init)∈C::update),empty),global1initdef,C::update.global1initdef},
{empty,(type(a1init)∈A::update),empty),
 a1init,A::update.thisinitdef},
 {empty,(type(a1init)∈C::update),empty),
 a1init,A::update.thisinitdef}
9. process-return-site( n, rdfe ) {
    /* n is the return site of a call node */
    if (first case and ecfi is empty)
        y = ordinary successor of n
    else
        y = exit node of innermost-enclosing-exception-block(n)
    add-to-soln-and-worklist-if-needed(y, {rdfe});
}

10. process-method-exit( n, rdfe ) {
    /* n is the exit node of the method n.method */
    if (first case and var is a local variable)
        /* no need to propagate to callers */
        return
    for each call site c in the current SCC
        that invokes n.method {
            x = y = z = ϕ
            x = back-bind( rdfe, c, n )
            /* back-bind returns
               [ecfi1,rc3,(var1,def1,(rc4,val1))]'s
               or [rc3,excp-obj1,throw-point]'s
               to which rdfe maps at c */
            if (first case and def is a program point)
                /* i.e., the definition has been generated during
                   the life-time of n.method */
                for each dfe ∈ x
                    generate-defs-due-to-pot-aliases(c.method, dfe)
            z = x ∪ y
            add-to-soln-and-worklist-if-needed-I(c.successor, z)
        }
}

Figure 13: Return-site and method-exit-node transfer functions

E  process-return-site, process-method-exit and process-return-statement

back-bind( dfe, c, n ) uses the bindings in c.actual-to-uvv-bindings
and c.def-udef-bindings to instantiate the unknown initial values
and the unknown initial defs in dfe with their values at c. back-bind returns
the set of DUA-dfelm results from the instantiations.

11. process-return-statement( n, rdfe ) {
    /* rdfe is dfelm-c [ecfi,rc1,(var,def,(rc2,val))] */
    if ( n represents return var )
        /* var is read in n */
        n.def-uses = n.def-uses ∪ [rc1,(var,def,n)]
    /* n.method is the method containing n */
    x = exit of n.method
    add-to-soln-and-worklist-if-needed-I(n, rdfe)
}

Figure 14: Return statement transfer function