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 $a1_{init}$ and $a2_{init}$, the unknown objects to which *a1* and *a2* point at the entry node of *method1*, are the same object, statement **4** modifies the *next* field of $a2_{init}$ in addition to the *next* field of $a1_{init}$. As a result, there is a def-use relationship between statements **4** and **6** if and only if $a1_{init}$ and $a2_{init}$ 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 $a4_{init}$ and $a5_{init}$ identical at the entry node of the public interface method *method2* (which will in turn make $a1_{init}$ and $a2_{init}$ 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 $a1_{init} \in \{A, B\}$, a p-use of r exists along the interprocedural edge e_1 from statement **5** to the entry node of *A.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 $a1_{init}$ is C, a p-use of r exists along the interprocedural edge e_2 from statement **5** to the entry node of *C.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., $e1_{init}$) in Figure 1 is caught by the catch statement at program point **16** if and only if the concrete type of $e1_{init}$ 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 $e1_{init}$ is *ET2* or a subtype of *ET2*. Similarly, if the exception is caught by the catch

[1] A method that can be invoked directly from outside the library.

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[2] 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[3]. It includes single inheritance, dynamic dispatch, recursive types, exceptions and pointer assignment statements with a single level of dereferencing[4] (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

[2] Points-to analysis [EGH94] traces value flow through variables of pointer or reference type.

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

[4] lhs = rhs, where lhs is of the form p or p.f1 and rhs is also of the form q or q.f2.

```
package L_example;
public class A {
  private A next;
  void update(){
    1:  Lib.global1 = new A();
    Lib.global3 = Lib.global1;}
  };
public class B extends A {};

public class C extends A {
  void update() {
    2:  Lib.global1 = new C();
    Lib.global3 = Lib.global1;
  };
};
```

```
/*** ET1, ET2, ET3 and ET4 are public classes ***/
class ET1 extends Exception { };  class ET2 extends ET1 { };
class ET3 extends ET2 { };        class ET4 extends ET1 { };
```

```
public class Lib {
public static A global1, global2, global3;
private static void method1(A a1, A a2, A a3) {
  A r, u;

  3:  r = a1;
  4:  a1.next = a3;
  5:  r.update();
  6:  global2 = a2.next;
  7:  u = global1; }

public static void method2( A a4, A a5, A a6 ) {
  /*** public interface method ***/
  A p, q;
  8:  method1(a4,a5,a6);
  9:  p = new B(); 10:  q = new C();
  11:  method1(p,q,q);
  12:  }

private static void method3 ( ET1 e1 ) {
  A u1;
  try {
      13:  global3 = new A();
      14:  throw e1;
    15:
  }
  16:catch ( ET2 excp ) {
    17:  u1 = global3;
  }
  18:  }

public static void method4( ET1 e2 ) {
  /*** public interface method ***/
  A u2,u3; ET1 e3,e4;
  if ( _ ) {
    19:  e3 = new ET4();
    try {
      20:  method3( e3 );
    }
    21:catch( ET4 excp1 ) {
      22:  u2 = global3;
    }
  }
  else {
    e4 = e2;
    try {
      23:  method3( e4 );
    }
    24:  catch( ET4 excp2 ) {
      25:  u2 = global3;
    }
    26:  catch( ET1 excp3 ) {
      27:  u3 = global3;
    }
  }
  28:  } }; /*** end of class Lib ***/
```

Figure 1: Library $L_{example}$

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 reference type. Any other kind of assignment statement is represented by a *non-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 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 $object_n$, a **heap-name**. For example, all objects created at program point **9** in Figure 1 are represented by $object_9$.

Unknown initial value(UIV): var_{init} 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, $var_{init}.next_{init}$ represents the unknown initial value to which *var.next* points, $var_{init}.next_{init}.next_{init}$ represents the unknown initial value to which *var.next.next* points, and so on. For example, in Figure 1 $a1_{init}$ represents the unknown initial object to which parameter *a1* points at the entry node of *method1*. When *method1* is invoked from statement **11**, $a1_{init}$ maps to $object_9$, while at statement **8** $a1_{init}$ maps to $a4_{init}$. 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 **11** $object_9$, an object of type B, maps to $a1_{init}$ 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* [Deu94] 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 ⇒ {Class | Proc}+

Class ⇒ [Protection] class ClassName
  [ extends (ClassName | Exception)]
  { {DataMember| Method}+ }

DataMember ⇒ [Protection] [static] Type FieldName ;

Type ⇒ ClassName | PrimitiveType
PrimitiveType ⇒ int | char | float | boolean

Method ⇒ Protection [static | virtual] [void | Type]
  MethodName ( [Param RestParam*] ) { Body }
Param ⇒ Type VarName
RestParam ⇒ , Param

Proc ⇒ {void | Type} ProcName ( [Param RestParam*] )
  { Body }

Body ⇒ Decls Stmnt+

Decls ⇒ Decl*
Decl ⇒ Type VarName ;

Stmnt ⇒ AssignmentStmnt | NewStmnt | Call | If |
  While| ReturnStmnt | Try | Throw | ;

ReturnStmnt ⇒ return VarName ;

AssignmentStmnt ⇒ Lhs = Rhs ;
Lhs ⇒ VarName | VarName→FieldName
Rhs ⇒ VarName | VarName→FieldName | 0 | Expr

Call ⇒ [VarName = ]
  {VarName→MethodName|MethodName
  |ClassName::MethodName|ProcName}
  ( [VarName RestVar*] ) ;

NewStmnt ⇒ VarName = new ClassName ( [VarName RestVar*] ) ;

RestVar ⇒ , VarName

If ⇒ if ( Expr ) {Stmnt+ } [else { Stmnt+ } ]

While ⇒ [Label :] while ( Expr ) { Stmnt+ }

Throw ⇒ throw VarName ;
Try ⇒ try { Stmnt+ } Catch*
Catch ⇒ catch ( ClassName * VarName ) { Stmnt+ }

VarName ⇒ Name
FieldName ⇒ Name
ProcName ⇒ Name
MethodName ⇒ Name
ClassName ⇒ Name
Label ⇒ Name
Protection ⇒ public | protected | private
```

Figure 2: Grammar for *RL*

terface initial value. For example, in Figure 1 $a4_{init}$ 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 a unknown initial value, and it can be of any type. For example in Figure 1 at statement **8** $global2_{initdef}$ 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, object*$\rangle$, 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 *object* is an unknown initial value or heap-name.[5] *object* 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, def-point, 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 *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 *def-point* and this definition of *loc* is used at *use-point*. In Figure 1, $\langle global3,\mathbf{13},\mathbf{17}\rangle$ represents the def-use relationship due to the definition and use of *global*3 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 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].[6] 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 I

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 I-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

[5] We refer to *pointer* and *reference* types interchangeably in this paper.

[6] Function pointer call targets in initial call graph can be approximated by those functions whose addresses have been stored and whose signatures match with the type of the called function.

pointer summary transfer functions of methods in the same SCC have cyclic dependences, so they are computed simultaneously by fixed point iteration. In contrast, the pointer summary transfer functions of methods in different SCC's have hierarchical dependences (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 $a2_{init}$ if and only if *a1* and *a2* are aliased at the entry of *method1*, and hence I-pta infers that on the top of statement **5**, $a2_{init}.next$ points to $a3_{init}$ under the condition that $a1_{init}$ and $a2_{init}$ 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 $a1_{init}$, and hence I-pta infers that on the top of statement **6**, *global1* points to $object_1$ under the condition that the concrete type of $a1_{init} \in \{A,B\}$, and *global1* points to $object_2$ under the condition that the concrete type of $a1_{init} \in \{C\}$. 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**, $a1_{init}$ maps to $object_9$ and $a2_{init}$ maps to $object_{10}$. As a result, the condition saying $a1_{init}$ is same as $a2_{init}$ 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** $object_9.next$ points to $object_{10}$ and $object_{10}.next$ does not point to $object_{10}$, instead $object_{10}.next$ points to *null* (i.e., $object_{10}.next$ 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 $a1_{init}$ and $a2_{init}$, but it does not use the *next* field of $a3_{init}$. As a result, although statement **4** can modify $a3_{init}.next$ as well as $a2_{init}.next$ due to aliasing of $a1_{init}$ with $a2_{init}$ and $a3_{init}$, only a data-flow element representing conditional modification of $a2_{init}.next$ 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 dependences, 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 dependences (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 $a2_{init}.next$ at statement **4** and later its use at statement **6** is computed dependent on the condition that $a1_{init}$ and $a2_{init}$ 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 $a1_{init} \in \{A,B\}$, 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 $a1_{init} \in \{C\}$. 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** $a1_{init}$ maps to $object_9$. As a result, the condition that the concrete type of $a1_{init} \in \{A,B\}$ 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 $a2_{init}.next$ 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 $a2_{init}.next$, computed at statement **6**, is parametrized by the unknown initial def of $a2_{init}.next$ (i.e., $a2_{init}.next_{initdef}$).

3.3 Phase II

In this phase Def-Use-Algo traverses the SCC-DAG in a topological order (top-down) and performs the two subphases 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 $object_9$ and $object_{10}$ from call site **11** to the entry node of *method1* as a concrete values of $a1_{init}$ and $a2_{init}$ 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 $a4_{init}$ to the entry of *method1* as a concrete value of $a1_{init}$.

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 $object_{10}$ is initialized (defined), and hence at call site **11**, II-dua propagates program point **10** to the entry node of *method1* as a concrete value of $a2_{init}.next_{initdef}$, the unknown initial def of $a2_{init}.next$. 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 $a5_{init}.next_{initdef}$ to the entry node of *method1* as a concrete value of the unknown initial def $a2_{init}.next_{initdef}$.

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 *global1* is computed dependent on the condition that the concrete type of $a1_{init} \in \{C\}$. The interface initial value $a4_{init}$ (bound at call site **8**) and $object_9$ (bound at call site **11**) are the concrete values of $a1_{init}$ computed during II-pta. When $a1_{init}$ is instantiated with $object_9$ during Phase III, the condition asserting concrete type of $a1_{init} \in \{C\}$ evaluates to false. On the other hand, when $a1_{init}$ is instantiated with $a4_{init}$ during Phase III, the same condition evaluates to true because Def-Use-Algo makes conservative, worst-case assumptions about the interface initial value $a4_{init}$. Hence the final def-use solution contains the def-use relationship resulting from the definition and use of *global1* at statements **2** and **7** respectively, and this def-use is conditioned on the fact that the concrete type of the interface initial value $a4_{init} \in \{C\}$. Similarly, during I-dua the def-use relationship between the statements **4** and **6** due the definition of $a2_{init}.next$ at statement **4** and later its use at statement **6** is computed dependent on the condition that $a1_{init}$ and $a2_{init}$ are the same object at the entry of *method1*. In addition to the concrete values of $a1_{init}$ mentioned above, II-pta computes the interface initial value $a5_{init}$ (bound at call site **8**) and $object_{10}$ (bound at call site **11**) as the concrete values of $a2_{init}$. When $a1_{init}$ and $a2_{init}$ are instantiated with their concrete values in Phase III, the condition that $a1_{init}$ and $a2_{init}$ are the same object can evaluate to true only for the case in which $a1_{init}$ is instantiated with $a4_{init}$ and $a2_{init}$ is instantiated with $a5_{init}$ (recall that Def-Use-Algo makes conservative, worst-case assumption that the interface initial values $a4_{init}$ and $a5_{init}$ 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 $a5_{init}.next$ at statements **4** and **6** respectively, and this def-use is conditioned on the fact that $a4_{init}$ and $a5_{init}$ are the same object at the entry node of the public interface method *method2*.

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

1. `There are two kinds of` **PTA-dfelms**:
dfelm-a. [*ECFInfo*, *relevant context*, *points-to*]
dfelm-b. [*relevant context*, *exception object*]

2. `There are five kinds of` **DUA-dfelms**:
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. [*relevant context*, ⟨*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 domains (e.g., recursive descent parsing), 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.

4 Phase I data-flow elements

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

PTA-dfelms are used for points-to analysis and are computed during subphase I-pta. *PTA-dfelms* represent (a) values of pointer variables and (b) exception objects. *dfelms*-**a** are used for propagating values of locations of pointer type. *dfelms*-**b** are used for propagating exception objects from throw statements to corresponding catch statements (if any). The propagation of exception objects require 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 *dfelms*-**a**.

DUA-dfelms are used for def-use analysis and are computed during subphase I-dua. *DUA-dfelms* represent variable definitions and def-use relationships. *DUA-dfelms* are computed using *PTA-dfelms* because computation of def-use relationships require the points-to solution. *dfelms*-**e**, *dfelms*-**f** and *dfelms*-**g** are needed because they represent three different kinds of def-use relationships described in Section 1. *dfelms*-**c** are needed for propagating variable definitions to their use points, where they are used for computing *dfelms*-**e** and *dfelms*-**g** depending upon whether the use is in a non-test expression or in a test expression. *dfelms*-**d** are needed for propagating exception objects from throw statements to catch statements, where they are used for computing *dfelms*-**f** . *dfelms*-**b** cannot be used for this purpose because they do not ecode 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 control-flow information* stores the signature of an uncaught exception (if any) along paths through which a *PTA-dfelm* or *DUA-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:

- *excp-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 $e1_{init}$ and hence when the *dfelm*-**a** representing the value of *global3* is propagated across statement **14**, $e1_{init}$ becomes the *ECFInfo* of the new *dfelm*-**a** propagated to statement **15**. The reason for storing an unknown initial value *uiv* as the signature of the exception when *uiv* is thrown as the exception object is that the concrete type of the exception can be the declared type of *uiv* or any subtype of the declared type. The concrete type of the exception will be determined using concrete values of *uiv*. For example, in Figure 1 when *method3* 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* $e1_{init}$ 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:

$(uiv_1 \textbf{ eq } uiv_2)$

and each **potential non-alias** has the form:

$(uiv_1 \textbf{ neq } uiv_2)$,

where uiv_1 and uiv_2 are unknown initial values. For example, in Figure 1, at the bottom of statement **4** the points-to $\langle a2_{init}.next, a3_{init}\rangle$ is computed conditioned on the potential alias $(a1_{init} \textbf{ eq } a2_{init})$, while the the points-to $\langle a2_{init}.next, a2_{init}\rangle$ is computed conditioned on the potential non-alias $(a1_{init} \textbf{ neq } a2_{init})$.

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:

$(type(uiv) \in T{::}x)$,

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, object*$_1$⟩ is computed conditioned on the type constraint*(type(a*1_{init}*)∈A::update)*, while the points-to ⟨*global1, object*$_2$⟩ is computed conditioned on the type constraint*(type(a*1_{init}*)∈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. $(type(uiv) \leq T)$ or
2. $(type(uiv) \not\leq T)$.

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, object*$_{13}$⟩ is computed conditioned on the exception context *(type(e*1_{init}*)* $\leq$ *ET2)*, because the points-to ⟨*global3, object*$_{13}$⟩ 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 *(type(e*1_{init}*)* $\leq$ *ET2)*. In contrast, at statement **18** the points-to ⟨*u1, null*⟩ is computed conditioned on the exception context *(type(e*1_{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 *(type(e*1_{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** [*empty*,⟨*(a*1_{init} **eq** *a*2_{init}*),empty,empty*⟩,⟨*a*2_{init}*.next,a*3_{init}⟩] is propagated from the exit node of *method1* to the return node of call site **8**, it is translated into
[*empty*,⟨*(a*4_{init} **eq** *a*5_{init}*),empty,empty*⟩,⟨*a*5_{init}*.next,a*6_{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 *(a*1_{init} **eq** *a*2_{init}*)* translates to *(object*$_9$ **eq** *object*$_{10}$*)*, which is false.

We will use **Empty-context** to represent the relevant context ⟨*empty, empty, empty*⟩.

dfelm-b: A *dfelm*-**b** is used for propagating an exception object. *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 *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** [*Empty-context* ,e1_{init}] to statement **15**, the exit of the try statement. From statement **15**, the *dfelm*-**b** [⟨*empty, empty, (type(e*1_{init}*)* $\leq$ *ET2)* ⟩,e1_{init}] is propagated to statement **16** because the exception is caught by the catch statement under the type constraint*(type(e*1_{init}*)* $\leq$ *ET2)*. At statement **16**, the above *dfelm*-**b** determines the value of *excp*, the parameter of the catch statement. Similarly, from statement **15** the *dfelm*-**b** [⟨*empty, empty, (type(e*1_{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(e*1_{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-pta.

dfelm-c: *dfelms*-**c** are used for propagating variable definitions to their use points. Each *dfelm*-**c** has the form [*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 *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, [*empty*,⟨*(a*1_{init} **eq** *a*2_{init}*),empty,empty*⟩,⟨*a*2_{init}*.next,4,*⟨*Empty-context,a*3_{init}⟩⟩] represents the potential definition of *a*2_{init}*.next* at statement **4** in Figure 1. In this case, *relevant context-1* is ⟨*(a*1_{init} **eq** *a*2_{init}*),empty,empty*⟩ because *a*2_{init}*.next* (the *var*) is defined at statement **4** under this context; in contrast, *relevant context-2* is *Empty-context* because the right-hand-side of the assignment statement (*a3*) has the value *a*3_{init} in all contexts. Similarly, [*empty*,*Empty-context*,⟨*global2,6,*⟨⟨*(a*1_{init} **eq** *a*2_{init}*),empty,empty*⟩*,a*3_{init}⟩⟩] represents the definition of *global2* at statement **6** in Figure 1. In this case, *relevant context-1* is *Empty-context* because *global2* (the *var*) is defined at statement **6** in all contexts, and *relevant context-2* is ⟨*(a*1_{init} **eq** *a*2_{init}*),empty,empty*⟩ because the right-hand-side of the assignment statement (*a*2_{init}*.next*) has the value *a*3_{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 [*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, [*Empty-*

context,$e1_{init}$,*14*] represents the exception thrown at statement **14** in Figure 1. Here *relevant context* is *Empty-context* because *e1* points to $e1_{init}$ at statement **14** (and hence $e1_{init}$ 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 parametrized in terms of unknown initial defs 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** [⟨($a1_{init}$ **eq** $a2_{init}$),*empty*,*empty*⟩,⟨$a2_{init}$*.next*,*4*,*6*⟩] represents the def-use of $a2_{init}$*.next* at statements **4** and **6** in Figure 1. The *relevant context* is ($a1_{init}$ **eq** $a2_{init}$) because $a2_{init}$*.next* is defined at statement **4** only in those calling contexts of *method1* in which *a1* and *a2* are aliased. Similarly, the *dfelm*-**e** [⟨($a1_{init}$ **neq** $a2_{init}$),*empty*,*empty*⟩,⟨$a2_{init}$*.next*,$a2_{init}$*.next*$_{initdef}$,*6*⟩] represents the def-use of $a2_{init}$*.next* because of the definitions of $a2_{init}$*.next* reaching statement **6** from the entry of *method1*. The *relevant context* is ($a1_{init}$ **neq** $a2_{init}$) because there exists a definition-free path for $a2_{init}$*.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*,(*type*($e1_{init}$) ≤ *ET2*)⟩,⟨$e1_{init}$,*14*,*16*⟩] represents the def-use relationship between the throw statement **14** and the catch statement **16** because the exception object $e1_{init}$ thrown at statement **14** is caught at statement **16** only in those calling contexts of *method3* in which the concrete type of $e1_{init}$ is *ET2* or a subtype of *ET2*.

dfelm-g: These *DUA-dfelms* 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,
[⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*r*,*3*,⟨*5*,*A::update*⟩⟩] and
[⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,⟨*r*,*3*,⟨*5*,*C::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 (*type*($a1_{init}$)∈*A::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 (*type*($a1_{init}$)∈*C::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-dfelms*, while the following theorem is for both *PTA-dfelms* and *DUA-dfelms*):

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-dfelms* 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. [*rc*, ⟨*var, def-point, use-point*⟩], 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. [*rc*, ⟨*excp-object, throw-point, catch-point*⟩], 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. [*rc*, ⟨*var, def-point, use-edge*⟩], where either *n* is a condition node and *use-edge* is a decision edge out of *n* or

n is a dynamically dispatched call site and *use-edge* is an interprocedural edge between n and one of its targets. *def-point* and *var* are same as in 1. Elements of this form express p-uses.

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 *dfelm*-**e** [⟨*empty,(type(*$a1_{init}$*)*∈*A::update),empty*⟩,⟨*global1,1,7*⟩] is instantiated to [⟨*empty,(type(*$a4_{init}$*)*∈*A::update),empty*⟩,⟨*global1,1,7*⟩] because $a4_{init}$ is an interface initial value and it a possible concrete value of $a1_{init}$ (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[Lan92].

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 *(type(*$a4_{init}$*)*∈*A::update)* is relevant for data-flow-based unit testing of the public interface method *method2* of $L_{example}$ 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 $a6_{init}$ is not relevant for data-flow-based unit testing of *method2* because no type constraint involving $a6_{init}$ 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. *size-of(T)* $\leq$ *size-of(C)*;
2. for each $c \in C$, $\exists t \in T \ni t \Rightarrow c$; and
3. for each $t \in T$, $\exists c \in C \ni t \Rightarrow c$.

Let $\mathbf{RelevantContextCover_L}$ be the set of relevant contexts of the def-use relationships computed by Phase III for a library **L**. Let $\mathbf{RelevantContextCover_M}$ be the subset of $\mathbf{RelevantContextCover_L}$ 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 fields 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 $\mathbf{Cover_{M_t}}$ be the subset of $RelevantContextCover_M$ that is consistent with the specification of t. One strategy for the tester is to manually[7] enhance t using each element of $Cover_{M_t}$ and instantiate each enhanced template into a test case. This will result in *size-of(*$Cover_{M_t}$*)* test cases which can be executed separately. However, there is a more efficient alternative: first the tester can compute a reduced context cover $\mathbf{ReducedCover_{M_t}}$ of $Cover_{M_t}$ having as small a size as possible and then generate *size-of(*$ReducedCover_{M_t}$*)* test cases by enhancing t using the elements of $ReducedCover_{M_t}$. 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 specific initial 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 $Cover_{M_t}$, the following simple greedy algorithm finds a reduced context cover $ReducedCover_{M_t}$ such that each element of $ReducedCover_{M_t}$ is consistent with t. The algorithm may not find the minimal cover in some cases.

```
ReducedCover_Mt = φ
for each s ∈ Cover_Mt
  if s contains a representative unknown initial value
    ReducedCover_Mt = ReducedCover_Mt ∪ {s}
  else
    if (∃ r ∈ ReducedCover_Mt such that r ∧ s can be
        satisfied and r ∧ s is consistent with t)
      new-el = r ∧ s
      ReducedCover_Mt = ReducedCover_Mt - {r} ∪ {new-el}
    else
      ReducedCover_Mt = ReducedCover_Mt ∪ {s}
```

For example, $RelevantContextCover_{L_{example}}$ for the library $L_{example}$ given in Figure 1 is **{** ⟨*empty*⟩, ⟨*(*$a4_{init}$ **eq** $a5_{init}$*)*⟩, ⟨*(*$a4_{init}$ **neq** $a5_{init}$*)*⟩, ⟨*(type(*$a4_{init}$*)*∈*A::update)*⟩, ⟨*(type(*$a4_{init}$*)*∈*C::update)*⟩, ⟨*(type(*$e2_{init}$*)* $\leq$ *ET2)*⟩, ⟨*(type(*$e2_{init}$*)* $\leq$ *ET4)*⟩, ⟨*(type(*$e2_{init}$*)* $\not\leq$ *ET2)* ∧ *(type(*$e2_{init}$*)* $\not\leq$ *ET4)*⟩**}**. The relevant def-uses whose relevant contexts are the non-empty elements of $RelevantContextCover_{L_{example}}$ 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 $L_{example}$ where all pointer variables are left unspecified by the tester. $RelevantContextCover_{method2}$ is **{** ⟨*empty*⟩, ⟨*(*$a4_{init}$ **eq** $a5_{init}$*)*⟩, ⟨*(*$a4_{init}$ **neq** $a5_{init}$*)*⟩, ⟨*(type(*$a4_{init}$*)*∈*A::update)*⟩, ⟨*(type(*$a4_{init}$*)*∈*C::update)*⟩**}** and

[7] It is possible to automate this step to a great extent and we plan to investigate this in future.

$Cover_{method2_{t1}}$ is the same as $RelevantContextCover_{method2}$. The reduced context cover $ReducedCover_{method2_{t1}}$ computed by the algorithm given above is
{ ⟨*(*$a4_{init}$ **eq** $a5_{init}$*)* ∧ *(type(*$a4_{init}$*)*∈*A::update)*⟩,
⟨*(*$a4_{init}$ **neq** $a5_{init}$*)* ∧ *(type(*$a4_{init}$*)*∈*C::update)*⟩**}**.

Consider the test case that can be generated from $t1$ after enhancing $t1$ with the first condition of the reduced cover.

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

Similarly, the following test case can be generated from $t1$ after enhancing $t1$ with the second condition.

```
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 *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 *ECFInfo* is still enough because the stack is implicitly maintained at call sites by storing *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 *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 *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 [⟨($a1_{init}$ **eq** $a2_{init}$),*empty*,*empty*⟩,⟨$a2_{init}$.*next*,*4*,*6*⟩]
6:2 [⟨($a1_{init}$ **neq** $a2_{init}$),*empty*,*empty*⟩,⟨$a2_{init}$.*next*,$a2_{init}$.$next_{initdef}$,*6*⟩]

7:1 [⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*global1*,*1*,*7*⟩]
7:2 [⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,⟨*global1*,*2*,*7*⟩]

17:1 [⟨*empty*,*empty*,(*type*($e1_{init}$) $\leq$ *ET2*)⟩,⟨*global3*,*13*,*17*⟩]

25:1 [⟨*empty*,*empty*,(*type*($e2_{init}$) $\leq$ *ET4*)⟩,⟨*global3*,*13*,*25*⟩]

27:1 [⟨*empty*,*empty*,(*type*($e2_{init}$) $\not\leq$ *ET2*) $\wedge$ (*type*($e2_{init}$) $\not\leq$ *ET4*)⟩,⟨*global3*,*13*,*27*⟩]

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

`/*** interface initial values` $a4_{init}$ `and` $a5_{init}$ `are respectively concrete values of` $a1_{init}$ `and` $a2_{init}$ `at call site` **8**`,`
`and interface initial def` $a5_{init}.next_{initdef}$ `is the concrete value of` $a2_{init}.next_{initdef}$ `at call site` **8** `***/`
6:1 [⟨($a4_{init}$ **eq** $a5_{init}$),*empty*,*empty*⟩,⟨$a5_{init}$.*next*,*4*,*6*⟩]
6:2 [⟨($a4_{init}$ **neq** $a5_{init}$),*empty*,*empty*⟩,⟨$a5_{init}$.*next*,$a5_{init}$.$next_{initdef}$,*6*⟩]

7:1 [⟨*empty*,(*type*($a4_{init}$)∈*A::update*),*empty*⟩,⟨*global1*,*1*,*7*⟩]
7:2 [⟨*empty*,(*type*($a4_{init}$)∈*C::update*),*empty*⟩,⟨*global1*,*2*,*7*⟩]

`/*** interface initial value` $e2_{init}$ `is the concrete value of` $e1_{init}$ `at call site` **23** `***/`
17:1 [⟨*empty*,*empty*,(*type*($e2_{init}$) $\leq$ *ET2*)⟩,⟨*global3*,*13*,*17*⟩]

25:1 [⟨*empty*,*empty*,(*type*($e2_{init}$) $\leq$ *ET4*)⟩,⟨*global3*,*13*,*25*⟩]

27:1 [⟨*empty*,*empty*,(*type*($e2_{init}$) $\not\leq$ *ET2*) $\wedge$ (*type*($e2_{init}$) $\not\leq$ *ET4*)⟩,⟨*global3*,*13*,*27*⟩]

Figure 5: Part of Phase III def-use solution

References

[Bei90] Boris Beizer. *Software Testing Techniques.* New York: Van Nostrand Reinhold, 1990.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[Cha99] Ramkrishna Chatterjee. Modular dataflow analysis of statically typed object-oriented languages, phd thesis. Technical Report DCS-TR-406, Dept of CS, Rutgers University, 1999.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* The MIT Press and McGraw-Hill Book Company, 1992.

[CRL98] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Relevant context inference. Technical Report DCS-TR-360, Dept of CS, Rutgers University, July 1998.

[CRL99] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Relevant context inference. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1999.

[Deu94] A. Deutsch. Interprocedural may alias for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[DGS96] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of the International Conference on Software Engineering (ICSE'96)*, March 1996.

[DMM96] Amer Diwan, J.Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 242–256, 1994.

[FO76] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8:305–330, September 1976.

[FW88] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[FW93] P. Frankel and S. Weiss. An experimental comparison of the effectiveness of branch testing with data-flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[HL91] J.R. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the 4th Symposium on Testing, Analysis, and Verification*, pages 87–97, 1991.

[HR94] Mary Jean Harrold and Gregg Rothermel. Performing dataflow testing on classes. In *Proceedings of the second Symposium on the Foundations of Software Engineering*, December 1994.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[Lan92] W. A. Landi. Undecidability of static analysis. *acm Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[LCS89] D. Richardson L.A. Clarke, A. Podgurski and S. Seil. A comparison of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[LK83] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9:347–354, May 1983.

[LR91] W.A. Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1991.

[LR92] W.A. Landi and Barbara G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[Ost90] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[OW88] T. J. Ostrand and E. Weyuker. Using data flow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, September 1988.

[OW91] T. J. Ostrand and E. Weyuker. Data flow based test adequecy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991. Victoria, B.C., Canada.

[PFW85] S. Weiss P.G. Frankl and E.J. Weyuker. Asset: A system to select and evaluate tests. In *In Proceedings of the IEEE Conference on Software Tools*, pages 72–79, April 1985.

[PLR94] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[RR01] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *to appear in Proceedings of the International Conference on Compiler Construction*, April 2001. also avaiable as Rutgers Department of Computer Science Technical Report DCS-TR-423.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William A. Landi. Data-flow analysis of program fragments. In *Proceedings of Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, LNCS 1687, pages 235–252, September 1999.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[SH99] Saurabh Sinha and Mary Jean Harrold. Criteria for testing exception-handling constructs in java programs. In *Proceedings of the International Conference on Software Maintenance*, August 1999.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[SP00] Amie Souter and Lori L. Pollock. Omen: A strategy for testing object-oriented software. In *Proceedings of ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis*, pages 49–59, August 2000.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Wey94] E. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, September 1994.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 1–12, 1995.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programming, Systems and Applications*, 1999.


```
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 $L_{example}$ 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-dfelms*, 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-uiv 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 defs 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-

```
initialize-worklist-I-dua( ) {
  worklist = empty

  for each method entry node n in the current SCC
    1:  propagate-unknown-init-defs-from-entry-node( n )

  for each reachable assignment node n in the current SCC
    2:  propagate-defs-from-assignment-node( n )

  for each reachable throw node n in the current SCC
    3:  propagate-exception-objects-from-throw-node( n )

  for each reachable object creation site n in the
      current SCC {
    4:  propagate-defs-created-at-an-object-creation-site(n)
  }

  for each reachable call site n in the current SCC
    if (the result of the call is stored in a variable)
      Let n.lhs be the variable p
      for each [ecfi,rc,⟨p,val⟩] ∈ n.successor.reaching-PTA-dfelms
        y = [ecfi,Empty-context,⟨p,n,⟨rc,val⟩⟩]
        add-to-soln-and-worklist-if-needed-I-dua(n.successor.successor,
                               {y})
    for each method m invocable from n
      if (m is not in the current SCC)
        5:back-bind-using-def-use-summary-transfer-
                                           function(m,n)
}
```

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*,⟨$a1_{init}$*.next*,*4*,⟨*Empty-context*,$a3_{init}$⟩⟩], and this def implies the potential def
[*empty*,⟨($a1_{init}$ **eq** $a2_{init}$),*empty*,*empty*⟩,⟨$a2_{init}$*.next*,*4*,⟨*Empty-context*,$a3_{init}$⟩⟩], both of which are added to the worklist.

propagate-exception-objects-from-throw-node (Figure 7) initiates the propagation of *dfelms*-**d** 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, $e1_{init}$; hence, [*Empty-context*,$e1_{init}$,*14*] is propagated to the exit node of *innermost-enclosing-exception-block(14)*, 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 *dfelms*-**c** representing these definitions to the *worklist*.

back-bind-using-def-use-summary-transfer-function (Figure 7) instantiates the *DUA-dfelms* at the exit node of a non-same-SCC method *m* using actual-to-uiv bindings 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-uiv bindings are computed by *phase-I-pta*. For example, [*empty*,*Empty-context*,⟨$a1_{init}$*.next*,*4*,⟨*Empty-context*,$a3_{init}$⟩⟩] 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*,⟨$a4_{init}$*.next*,*4*,⟨*Empty-context*,$a6_{init}$⟩⟩],

```
process-worklist-I-dua( ) {
while worklist is not empty {
  wl-node = delete from work-list
  /*** wl-node = (node,dfe) ***/
  n = wl-node.node
  rdfe = wl-node.dfe

  1:  if ( n is an assignment node )
        process-assignment( n, rdfe )

  2:  if ( n is a condition node )
        /*** i.e., n represents the test expression of
        *** if or while ***/
      process-condition( n, rdfe )

  3:  if ( n is a object creation node )
        /*** e.g., p = new A() ***/
        process-new( n, rdfe )

  4:  if ( n is a call node )
        process-call( n, rdfe )

  5:  if ( n is a throw node )
        process-throw( n, rdfe )

  6:  if ( n is the exit node of a try block )
        process-try-exit( n, rdfe )

  7:  if ( n is the entry of a catch statement)
        process-catch( n, rdfe )

  8:  if ( n is the exit node of a catch statement )
        process-catch-exit( n, rdfe )

  9:  if ( n is a return site )
        /*** n is the successor of a call node ***/
        process-return-site( n, rdfe )

  10:  if ( n is a method exit node )
        process-method-exit( n, rdfe )

  11:  if ( n represents a return statement )
        process-return-statement( n, rdfe )
} }
```

Figure 8: process worklist for Phase I-dua

which is added to the worklist.

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-dfelms-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*,
⟨$a2_{init}$*.next*,$a2_{init}$*.next*$_{initdef}$,⟨*Empty-context*,$a2_{init}$*.next*$_{init}$⟩⟩]
at statement **4** in Figure 1, *generate-dfelms-due-to-pot-non-aliases* generates the set of *dfelms*-**c**
{[*empty*,⟨($a1_{init}$ *neq* $a2_{init}$),*empty*,*empty*⟩,
⟨$a2_{init}$*.next*,$a2_{init}$*.next*$_{initdef}$, ⟨*Empty-context*,$a2_{init}$*.next*$_{init}$⟩⟩]**}**.

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*,$a1_{init}$⟩⟩] at statement **5** in Figure 1, *val* is $a1_{init}$ and the possible targets with $a1_{init}$ as receiver are *A::update* and *C::update*. As a result, the *dfelms*-**g**
[⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,⟨*r*,*3*,⟨*5*,*A::update*⟩⟩]
and
[⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,⟨*r*,*3*,⟨*5*,*C::update*⟩⟩]
are added to **5**.*def-uses*.

```
/*** rdfe is dfelm-c [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
1. process-assignment( n, rdfe ) {
  if ( var is read in n )
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,n⟩]

  if ( n must update var )
    /*** rdfe is killed by n ***/
    return
  else
    x = generate-dfelms-due-to-pot-non-aliases(n, rdfe)
    add-to-soln-and-worklist-if-needed-I(n.successor, x)
}
```

```
2. process-condition( n, rdfe ) {
  if ( var is read in n ) {
    /*** p uses ***/
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,(n,true-successor)⟩]
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,(n,false-successor)⟩]
  }

  for each successor succ of n
    add-to-soln-and-worklist-if-needed-I(succ,{rdfe})
}
```

```
3. process-new( n, rdfe ) {
  if ( var is read in n )
    n.def-uses = n.def-uses ∪ [rc1,⟨var,def,n⟩]
  if ( var is n.left-hand-side ) {
    /*** rdfe is killed, need not propagate further ***/
    return
  }
  else
    add-to-soln-and-worklist-if-needed-I(n.successor,{rdfe})
}
```

```
4. process-call( n, rdfe ) {
  if ( var is an actual )
    n.def-uses = n.def-uses ∪ {[rc1,⟨var,def,n⟩]}

  if ( var is the receiver variable )
    /*** e.g., in p→foo(), p is the receiver variable ***/
    a:  compute-p-uses-due-to-dynamic-dispatch(n, rdfe)

  b:  new-def-uidef-bindings =
      compute-new-bindings-of-defs-and-u-i-defs(n, rdfe)
  n.def-uidef-bindings = n.def-uidef-bindings ∪
                         new-def-uidef-bindings

  if (var is not n.left-hand-side)
    add-to-soln-and-worklist-if-needed-I(n.successor,{rdfe})
    for each exception object excp-object thrown
        by a method invocable from n
      for each relevant context rc3 under which excp-object
                         is thrown
        rc4 = rc1 ∧ rc3
        new-dfe = [get-ecfi(excp-object),rc4,⟨var,def,⟨rc2,val⟩⟩]
        add-to-soln-and-worklist-if-needed-I(n.successor,
                         {new-dfe})
  /*** else rdfe is killed as the result of the call is
      stored in var ***/
}
```

```
add-to-soln-and-worklist-if-needed-I( node, dfelms ) {
  for each dfe ∈ dfelms
  if dfe ∉ node.reaching-defs
    node.reaching-defs = node.reaching-defs ∪ {dfe}
    wl-node = new worklist node (node, dfe)
    add wl-node to worklist }
```

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*,⟨*var,def*⟩,*uidef*]; 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*⟩,$a2_{init}.next_{initdef}$] ∈ **11**.*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 *excp-object* is a heap-name, *get-ecfi(excp-object)* returns *typeof(excp-object)*. Otherwise *excp-object* is an unknown initial value and *get-efci(excp-object)* returns *excp-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.

5. process-throw (Figure 10) uses the points-to solution computed by Phase I-pta at the *throw* statement to determine (1) the exception objects thrown and (2) the relevant contexts in which these objects are thrown. It stores the signature of the exception in the *ECFInfo* of the generated *dfelm*-**c**. The *ECFInfo* determines the future propagation of the generated *dfelm*-**c** from the exit node of *innermost-enclosing-exception-block(n)*. For example, at statement **14** in Figure 1, there is only one *eo*, $e1_{init}$, and the corresponding *rc3* is *Empty-context*. As a result, when the *rdfe* [*empty*,*Empty-context*,⟨*global3,13,*⟨*Empty-context,*$object_{13}$⟩⟩] reaches statement **14** (from statement **13**), the *dfelm*-**c** [$e1_{init}$,*Empty-context*,⟨*global3,13,*⟨*Empty-context,*$object_{13}$⟩⟩] is generated.

6. process-try-exit (Figure 10): There are two cases: either (1) *rdfe* represents a variable definition or (2) *rdfe* represents an exception object. If *rdfe* represents a variable definition and it flows to n along a path without any uncaught exception (i.e., *ecfi* is *empty*), *process-try-exit* propagates *rdfe* to the successor of n, i.e., the statement following the *try-catch* construct. Otherwise, it propagates *rdfe* to the entry nodes of the catch statements that are associated with the try statement and that can catch the exception represented by *ecfi* or *excp-obj*. If the exception can escape all of the catch statements associated with the try statement, *process-try-exit* propagates a new *DUA-dfelm* to the exit node of *innermost-enclosing-exception-block(n)*. *get-escape-tcs* returns a conjunction of type constraints which say under what conditions the exception can escape all of the catch statements associated with the try statement. For example, (*first case*) let *rdfe* be the *dfelm*-**c** [$e1_{init}$,*Empty-context*,⟨*global3,13,*⟨*Empty-context,*$object_{13}$⟩⟩] at program point

```
5. process-throw( n, rdfe ) {
/*** rdfe is dfelm-c [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
new-dfelms-c = φ
if ( n represents throw var )
  n.def-uses = n.def-uses ∪ [rc1,⟨var,def,n⟩]
for each exception object eo thrown by n
  for each relevant context rc3 under which eo is
          thrown by n
    rc4 = rc1 ∧ rc3
    /*** STORE EXCEPTION SIGNATURE ***/
    if ( eo is a heap-name )
      new-dfelms-c = new-dfelms-c ∪
                       {[typeof(eo),rc4,⟨var,def,⟨rc2,val⟩⟩]}
    else
      /*** eo is an unknown initial value ***/
      new-dfelms-c = new-dfelms-c ∪
                       {[eo,rc4,⟨var,def,⟨rc2,val⟩⟩]}

succ = exit node of innermost-enclosing-exception-block(n)

add-to-soln-and-worklist-if-needed-I(succ,new-dfelms-c)
}
```

```
6. PROCESS-TRY-EXIT( n, rdfe ) {
/*** first case:rdfe is dfelm-c [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
      or
second case:
      rdfe is dfelm-d [rc1,excp-obj,throw-point]
***/

/*** successors contains the nodes to which rdfe
      needs to be propagated ***/
successors = φ

if (rdfe is [empty,rc1,⟨var,def,⟨rc2,val⟩⟩])
  /*** NORMAL TERMINATION ***/
  successors = successors ∪ {ordinary successor of n}
else {
  /*** TERMINATION DUE TO EXCEPTION ***/
  /*** either first case with non-empty ecfi
    or second case ***/

  /*** FIND APPROPRIATE CATCHES ****/
  for each catch c associated with n that can catch
      the exception represented by ecfi or excp-obj
    successors = successors ∪ { c.entry }

  /*** Can the exception ESCAPE all the catches?  ***/
  if (the exception represented by ecfi or excp-obj
      can escape all the catches associated with n)
    a:  propagate-escaped-exception(n, rdfe)
}
for each succ ∈ successors
  add-to-soln-and-worklist-if-needed-I(succ,{rdfe})
}
```

```
6.a. propagate-escaped-exception(n, rdfe) }
  let rc1 be ⟨ac1, tc1, eotc1⟩
  if (first case:  rdfe represents a variable def)
    new-rc = ⟨ac1, tc1, eotc1 ∧ get-escape-tcs(n, ecfi)⟩
    new-DUA-dfelm = [ecfi,new-rc,⟨var,def,⟨rc2,val⟩⟩]
  else
    /*** second case:
      rdfe represents an exception object ***/
    new-rc = ⟨ac1, tc1, eotc1 ∧ get-escape-tcs(n, excp-obj)⟩
    new-DUA-dfelm = [new-rc,excp-obj,throw-point]

  succ = exit node of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I(succ, {new-DUA-dfelm})
}
```

Figure 10: Transfer functions for throw and try-exit

15 in Figure 1, the exit node of the try statement in *method3*. Since *typeof*($e1_{init}$) 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*($e1_{init}$)*) $\not\leq$ ET2)* and the *dfelm*-**c** [$e1_{init}$,⟨*empty,empty,(type*(e_{init})*) $\not\leq$ ET2)*⟩,⟨*global3,13,*⟨*Empty-context,object*$_{13}$⟩⟩] 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*(e_{init})*) $\not\leq$ 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 *excp-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,object*$_{13}$⟩⟩] and *dfelm*-**d** [*Empty-context*,$e1_{init}$,*14*]. In the first case, *ECFInfo* is $e1_{init}$ and hence *get-catch-tcs* returns *(type*($e1_{init}$)*) $\leq$ ET2)*. As a result, *new-dfe* is the *dfelm*-**c** [*empty*,⟨*empty,empty,(type*($e1_{init}$)*) $\leq$ ET2)*⟩,⟨*global3,13,*⟨*Empty-context,object*$_{13}$⟩⟩]. In the second case, *excp-object* is $e1_{init}$ and again *get-catch-tcs* returns *(type*($e1_{init}$)*) $\leq$ ET2)*. As a result, the *dfelm*-**f** [⟨*empty,empty,(type*($e1_{init}$)*) $\leq$ ET2)*⟩,⟨$e1_{init}$*,14,16*⟩] is added to the solution at program point **16** and *new-dfe* = [*empty*,⟨*empty,empty,(type*($e1_{init}$)*) $\leq$ 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,

```
/*** first case:  rdfe is dfelm-c [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
            or
second case:
      rdfe is dfelm-d [rc1,excp-obj,throw-point]
***/
/*** let rc1 be ⟨ac1, tc1, eotc1⟩***/
7. PROCESS-CATCH( n, rdfe ) {
/*** n is the entry node of a catch statement that
  catches any exception whose type is catch-type
  or a subtype of catch-type ***/

if (first case)
  /*** rdfe represents a variable definition ***/
  a:  new-dfe = CATCH-VARIABLE-DEF(n, rdfe)
else
  /*** second case:  rdfe represents an exception object ***/
  b:  new-dfe = CATCH-EXCEPTION-OBJECT(n, rdfe)

add-to-soln-and-worklist-if-needed-I(n.successor, {new-dfe})
}
```

```
7.a. CATCH-VARIABLE-DEF(n, rdfe) {
if (rdfe is [exp-type,rc1,⟨var,def,⟨rc2,val⟩⟩])
  /** process-try-exit has ensured that exp-type is either same
      as the catch-type or it is a subtype of the catch-type **/
  new-dfe = [empty,rc1,⟨var,def,⟨rc2,val⟩⟩]
else
  /*** ecfi is an unknown initial value uiv.
      process-try-exit has ensured that typeof(uiv)
      and the catch-type are compatible ***/
  new-rc = ⟨ac1, tc1, eotc1 ∧ get-catch-tcs(n, uiv)⟩
  new-dfe = [empty,new-rc,⟨var,def,⟨rc2,val⟩⟩]
return new-dfe }
```

```
7.b. CATCH-EXCEPTION-OBJECT(n, rdfe) {
if ( excp-obj is a heap-name )
  new-rc = rc1
else
  /*** excp-obj is an unknown initial value ***/
  new-rc = ⟨ac1, tc1, eotc1 ∧ get-catch-tcs(n, excp-obj)⟩

  n.def-uses = n.def-uses ∪ [new-rc,⟨excp-obj,throw-point,n⟩]

  /*** param is the parameter of the catch ***/
  new-dfe = [empty,new-rc,⟨param,n,⟨Empty-context,excp-obj⟩⟩]
return new-dfe }
```

```
8. process-catch-exit( n, rdfe ) {

if (first case and
      var is the parameter of the catch statement)
  /*** need not propagate further because the def is
      local to the catch ***/
  return

if ( first case and ecfi is empty )
  /*** normal termination ***/
  successor = ordinary successor of n
else
  /*** termination due to exceptions ***/
  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

and then incrementally expands this set during Phase II-pta. The initial set consists of all the public interface methods of a library (for a complete program the initial set consists of only *main*). When Def-Use-Algo visits each node of SCC-DAG in a topological order during Phase II-pta, it considers only those methods in the current SCC that have been marked reachable. Whenever Def-Use-Algo finds an unmarked method to be invocable from a call site in a reachable method, it marks the new method also as reachable.

Phase II-dua consists of *initialize-worklist-II-dua* and *process-worklist-II-dua*.

initialize-worklist-II-dua (Figure 12) initializes the worklist with bindings between unknown initial defs and their values at reachable call sites. It considers a binding only if the value of the unknown initial def is a program point or an interface initial def. Propagation through a binding in which the value of the unknown initial def is itself an unknown initial def that is not an interface initial def is done by *process-worklist-II-dua*. *instantiate-def-uidef-binding* instantiates the unknown initial values in the *rc* and *var* of a def-uidef binding with their concrete values computed at the entry node of the corresponding method during Phase II-pta. It returns those instantiations for which *rc* can evaluate to true, i.e., either (1) all the unknown initial values in *rc* are instantiated by heap-names and *rc* evaluates to true (*rc1* is *Empty-context* in this case), or (2) *rc* is instantiated into a relevant context *rc1* that involves only the interface initial values of a single public interface method and *rc1* can evaluate to true by making conservative, worst-case assumptions about these interface initial values. For example, consider call site **8** in public interface method *method2* in Figure 1. [*Empty-context*,$\langle a5_{init}.next, a5_{init}.next_{initdef}\rangle$,$a2_{init}.next_{initdef}$] $\in$ **8**.*def-uidef-bindings*, where $a5_{init}$ is an interface initial value and $a5_{init}.next_{initdef}$ is an interface initial def. As a result, *initialize-worklist-II-dua* propagates this binding as it is to the entry node of *method1*.

process-worklist-II-dua (Figure 12) iteratively propagates the values of unknown initial defs to the entry nodes of methods. *instantiate-rc-under-constraint(m, rc, var, s)* instantiates the unknown initial values in the *rc* with their concrete values computed at the entry node of *m* during Phase II-pta; if *var* is an unknown initial value occurring in *rc*, *instantiate-rc-under-constraint* uses only *s* as the concrete value of *var* during each instantiation (hence the suffix under-constraint). This is the reason why *var* is stored in a def-uidef-binding. *instantiate-rc-under-constraint* returns a set of instantiations of *rc* that can evaluate to true. Each instantiation in this set is either (1) *Empty-context*, meaning the *rc* evaluates to true for the instantiation, or (2) it is a relevant context that involves 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 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 ∈ m.reaching-defs
    wl-node = new worklist node (m, binding)
    add wl-node to worklist
  for each reachable call node c in m {
    for each def-uidef binding
        [rc,⟨var,def⟩,uidef1] ∈ c.def-uidef-bindings {
      if def is a program point or an interface initial def
        a:  result = instantiate-def-uidef-binding(m,
                   [rc,⟨var,def⟩,uidef1])
        // 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) {

```
result = φ
let binding be [rc,⟨var,def⟩,uidef1]
for each instantiation of UIV's in ⟨rc,var⟩
    with their concrete values computed at
    the entry node of method during Phase II-pta {
  if rc can be true after the instantiation
    let var1 be the value of var after the
        instantiation
    let rc1 be the value of rc after the
        instantiation
    result = result ∪ {[rc1,⟨var1,def⟩,uidef1]}
}
return result
}
```

---

2. process-worklist-II-dua() {

```
while worklist is not empty {

  wl-node = delete from work-list
  /*** wl-node = (method,binding) ***/
  m = wl-node.method
  binding = wl-node.binding
  let binding be [rc1,⟨s,def⟩,uidef1]

  for each reachable call node c in m
    for each def-uidef binding
        [rc2,⟨var,uidef1⟩,uidef2] ∈ c.def-uidef-bindings
      rc3 = rc1 ∧ rc2
      new-rcs =
        instantiate-rc-under-constraint(m, rc3, var, s)

      for each rc ∈ new-rcs
        reaching-def = [rc,⟨s,def⟩,uidef2]
        add-to-soln-and-worklist-
          if-needed-II(uidef2.method, {reaching-def})
} }
```

---

add-to-soln-and-worklist-if-needed-II(method, bindings) {

```
for each binding ∈ bindings
  if binding ∉ method.reaching-defs
    method.reaching-defs = method.reaching-defs ∪ {binding}
    if ( method is in the current SCC )
      wl-node = new worklist node (method, binding)
      add 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 UIV at n and u.f
      has been marked as read
    y=[empty,Empty-context,⟨u.f,u.f_initdef,⟨Empty-context,u.f_init⟩⟩]
    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,g_initdef,⟨Empty-context,g_init⟩⟩]
    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,n,⟨Empty-context,p_init⟩⟩]
    add-to-soln-and-worklist-if-needed-I(n.successor, {y})
}
```

```
2. propagate-defs-from-assignment-node( n ) {
  gen-dfelms-c = φ
  for each ⟨rc1, var⟩ in lhs-rc-loc-pairs
    for each ⟨rc2, val⟩ in rhs-rc-loc-pairs
      y = [empty,rc1,⟨var,n,⟨rc2,val⟩⟩]
      gen-dfelms-c = gen-dfelms-c ∪ {y}
      gen-dfelms-c = gen-dfelms-c ∪
        generate-defs-due-to-pot-aliases(n.method, y)

  add-to-soln-and-worklist-if-needed-I(n.successor, gen-dfelms-c)
}
2.a generate-defs-due-to-pot-aliases(method, dfe) {
  generated-dfes = φ
  let dfe be [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
  let rc1 be ⟨ac1, tc1, eotc1⟩
  if ( var is u.f and u is an UIV ) {
    for each UIV v at method.entry compatible with u
      if ( v.f has been marked as read ) {
        rc3 = ⟨ac1 ∧ (u eq v), tc1, eotc1⟩
        new-dfe = [ecfi,rc3,⟨v.f,def,⟨rc2,val⟩⟩]
        generated-dfes = generated-dfes ∪ {new-dfe}
      }
  }
  return generated-dfes }
```

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

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

```
5. back-bind-using-def-use-summary-transfer-function(m, c) {
  for each dfe ∈ m.exit-node.reaching-defs
    /*** dfe is dfelm-c [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] or
    dfe is dfelm-d [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
      [ecfi1,rc3,⟨var1,def,⟨rc4,val1⟩⟩]'s
      or [rc3,excp-obj1,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 p, 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 represent 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,*$a1_{init}.next$⟩} and *rhs-rc-loc-pairs* is {⟨*Empty-context,*$a3_{init}$⟩}. Similarly, at statement **6** in Figure 1, *lhs-rc-loc-pairs* is {⟨*Empty-context,global2*⟩} and *rhs-rc-loc-pairs* is
{⟨⟨($a1_{init}$ **eq** $a2_{init}$)*,empty,empty*⟩,$a3_{init}$⟩,
⟨⟨($a1_{init}$ **neq** $a2_{init}$)*,empty,empty*⟩,$a2_{init}.next_{init}$⟩}. This is because statement **4** modifies $a2_{init}.next$ if and only if $a1_{init}$ and $a2_{init}$ are the same object at the entry of *method1*, and it does not modify $a2_{init}.next$ if and only if $a1_{init}$ and $a2_{init}$ are not the same object at the entry of *method1*. As a result, Phase I-pta computes the *dfelms*-**a**
[*empty,*⟨($a1_{init}$ **eq** $a2_{init}$)*,empty,empty*⟩,⟨$a2_{init}.next$,$a3_{init}$⟩]
and
[*empty,*⟨($a1_{init}$ **neq** $a2_{init}$)*,empty,empty*⟩,
⟨$a2_{init}.next$,$a2_{init}.next_{init}$⟩] on the top of statement **6**.

generate-defs-due-to-pot-aliases generates *dfelms*-**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 the *dfelms*-**c** [*empty,* ⟨($a1_{init}$ *eq* $a2_{init}$)*,empty,empty*⟩, ⟨$a2_{init}.next$,*4*,⟨*Empty-context,* $a3_{init}$⟩⟩] when *method* is *method2* and *dfe* is
[*empty, Empty-context,* ⟨$a1_{init}.next$,*4*,⟨*Empty-context,* $a3_{init}$⟩⟩].

back-bind(dfe, c, n) uses the bindings in *c.actual-to-uiv-bindings* to instantiate the unknown initial values in *dfe* with their values at *c*. *back-bind* returns the set of *DUA-dfelms* resulting from the instantiations.

C generate-dfelms-due-to-pot-non-aliases

```
1.a generate-dfelms-due-to-pot-non-aliases(n, rdfe) {
/*** rdfe is dfelm-c [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
/*** n is an assignment node ***/
  generated-dfes = 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.f where u is an UIV and
      the declared type of u is compatible with A )
    for each ⟨rc3, loc⟩ in n.lhs-rc-loc-pairs
      if ( loc is v.f and v is an UIV compatible with u )
        rc4 = rc1 ∧ rc3 ∧ ⟨(v neq u), empty, empty⟩
        new-dfe = [ecfi,rc4,⟨var,def,⟨rc2,val⟩⟩]
        generated-dfes = generated-dfes ∪ {new-dfe}
      else
        generated-dfes = generated-dfes ∪ {rdfe}
  else
    generated-dfes = generated-dfes ∪ {rdfe}
  return generated-dfes
}
```

D Auxiliary functions for process-call

```
/*** rdfe is DUA-dfelm [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
4.a compute-p-uses-due-to-dynamic-dispatch(n, rdfe) {
/*** var is the 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 typeof(val) under which
          t is invoked with val as receiver
      new-rc = rc1 ∧ rc2 ∧ ⟨empty, cond, empty⟩
      /*** add a new dfelm-g to n.def-uses ***/
      n.def-uses = n.def-uses ∪ {[new-rc,⟨var,def,(n,t)⟩]}
  else /*** val is a heap-name ***/
    let t be the target invoked with val as receiver
    new-rc = rc1 ∧ rc2
    n.def-uses = n.def-uses ∪ {[new-rc,⟨var,def,(n,t)⟩]}
}
```

```
4.b compute-new-bindings-of-defs-and-u-i-defs(n, rdfe) {
new-def-uidef-bindings = φ
if ( var is a global ) {
  4.b.1: uidefs = unknown initial defs of var at the entry
        nodes of those targets in whose lifetime var is read
  for each y ∈ uidefs
    if ( [rc1,⟨var,def⟩,y] ∉ n.def-uidef-bindings )
      new-def-uidef-bindings =
        new-def-uidef-bindings ∪ {[rc1,⟨var,def⟩,y]}
}
if ( var is obj.f ) {
  /*** obj is a heap-name or an unknown initial value ***/
  uivs = uidefs = φ
  for each [rc3,obj,uiv1] ∈ n.actual-to-uiv-bindings
    uivs = uivs ∪ {⟨rc3,uiv1⟩}
  for each ⟨rc4,uiv2⟩ ∈ uivs
    if ( uiv2.f has been marked as read )
      uidefs = uidefs ∪ {⟨rc4,uiv2.f_initdef⟩}
  for each ⟨rc5,uiv3.f_initdef⟩ ∈ uidefs
    rc6 = rc1 ∧ rc5
    if ([rc6,⟨obj.f,def⟩,uiv3.f_initdef] ∉ n.def-uidef-bindings)
      new-def-uidef-bindings = new-def-uidef-bindings ∪
        {[rc6,⟨obj.f,def⟩,uiv3.f_initdef]}
}

return new-def-uidef-bindings }
```

4.b compute-new-bindings-of-defs-and-u-i-defs : For example, when *rdfe* is [*empty*,*Empty-context*, ⟨*global1*,*method1.global*$1_{initdef}$,⟨*Empty-context*,*global*1_{init}⟩⟩] at statement **5** in Figure 1, *uidefs* (at program point **4.b.1**) is {*A::update.global*$1_{initdef}$,*C::update.global*$1_{initdef}$}, i.e., *uidefs* contains the unknown initial defs of *global1* at the entry nodes of *A::update* and *C::update*. Phase I-pta stores in *n.actual-to-uiv-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-uiv-bindings* is
{ [⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩,*global*1_{init}, *A::update.global*1_{init}],
[⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩,*global*1_{init}, *C::update.global*1_{init}],
[⟨*empty*,(*type*($a1_{init}$)∈*A::update*),*empty*⟩, $a1_{init}$,*A::update.this*$_{init}$],
[⟨*empty*,(*type*($a1_{init}$)∈*C::update*),*empty*⟩, $a1_{init}$,*C::update.this*$_{init}$]}

```
/*** first case:  rdfe is dfelm-c [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
      or
  second case:
      rdfe is dfelm-d [rc1,excp-obj,throw-point]
***/
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-I(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
      y = y ∪
        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-uiv-bindings* and *c.def-uidef-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-dfelms* resulting 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