Data-flow-based Testing of Object-Oriented Libraries DCS-TR-382

Ramkrishna Chatterjee Barbara G. Ryder

Department of Computer Science
Rutgers University
Piscataway NJ 08855, USA
+1 732 445 2001
{ramkrish,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 associations in libraries written in object-oriented languages (e.g., C++ and Java) 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-uses 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.

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 testing criteria can be defined using def-use relationships. [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 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** 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 associations 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 initial 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. Consequently, even if all-path coverage is attained by a set of test cases for $L_{example}$, unless these test cases make $a1_{init}$ and $a2_{init}$ identical, the def-use relationship between statements **4** and **6** will not be executed. *This shows that due to potential def-use associations, 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. There is a p-use of r 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 p-use of r on the edge e_1 if and only if the concrete type of $a1_{init} \in \{A, B\}$. Similarly, there is a p-use of r 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 p-use of r on the edge e_2 if and only if the concrete type of $a1_{init}$ is C.

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 where 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. Consider statement **14** in Figure 1. The exception object thrown by statement **14** (i.e.,$e1_{init}$) 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 statement, there also exists a def-use relation-

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

ship 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 also need to be exercised.

The above discussion shows that the notion of def-use associations needs to be extended for O-O libraries. In this paper, we present (Sections 3, 4, 5 and 6) a new def-use algorithm that can compute the above kinds of def-use relationships (besides other ordinary def-use relationships) in libraries written in a substantial subset of C++/Java. We also show (Section 7) how the contexts associated with def-use relationships can be used for generating relevant test cases. Throughout the paper important points are indicated by ▶.

2 Definitions

This section presents many technical definitions needed to explain the algorithm and it also delimits the subsets of C++ and Java 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 features of C++ and Java (except threads). This allows us to simplify the presentation while demonstrating the interesting parts of our algorithm. *RL* is defined in Figure 2[2]. It includes single inheritance, dynamic dispatch, recursive types, exceptions and pointer assignment statements with a single level of dereferencing[3](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 C++. All exception types must be derived from the class Exception, 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 (array elements are mapped to a single representative element) and *finally* statements (as in Java).

If the algorithm is understood fully for *RL*, then handling of most of C++ and Java (except threads) requires handling of details but not any changes to the fundamental ideas of the algorithm. 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.

In addition, the algorithm can essentially 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 for which it is safe (with respect to def-use analysis) to consider them to be executed in 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 is not known statically.

Heap-name: Any static analysis technique needs to represent a potentially infinite number of heap-allocated, run-time objects with a finite number of names. Like many previous researchers [LR92], we represent all the run-time objects created at a program point n with the single name $object_n$, a **heap-name**.

Unknown initial value(UIV): We use var_{init} to represent the unknown initial object to which the global or parameter *var* of pointer type points at the entry node of a method. Similarly,

[2] *Expr* is any 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.

[3] lhs = rhs, where lhs is p or p→f1 and rhs is q or q→f2.

```
class A {                                  class C: public A {
  private:  A *next;                         private:  void update() {
  private:  void virtual update(){             2:  Lib::global1 = new C();
    1:  Lib::global1 = new A();                Lib::global3 = Lib::global1; }
    Lib::global3 = Lib::global1;}              friend Lib; }
  friend Lib; }
class B : public A {}
```

```
class ET1 :  public Exception { };  class ET2 :  public ET1 { };
class ET3 :  public ET2 { };        class ET4 :  public ET1 { };
```

```
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 ⇒ {Class | Proc}+

Class ⇒ class ClassName [: public (ClassName | Exception)]
  { {DataMember| Method}+ }

DataMember ⇒ Protection [static] Type FieldName ;

Type ⇒ ClassName * | PremitiveType
PremitiveType ⇒ int | char | float | bool

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*

$var_{init}.next_{init}$ represents the unknown initial value to which $var{\rightarrow}next$ points and $var_{init}.next_{init}.next_{init}$ represents the unknown initial value to which $var{\rightarrow}next{\rightarrow}next$ points. Obviously, in the presence of recursive types, the number of unknown initial values accessed in a method can be unbounded. To overcome this problem, unknown initial values are mapped into 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 [CRL98].

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

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.

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 they are the same class or they are related by a subtype-supertype relationship.

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

Def-Use Association: A def-use association is a triplet ⟨*loc, def-point, use-point*⟩, 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

1. 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* and
2. p is minimal: no prefix of p has the first property.

Exception Block: An *exception block* is the body of a try, catch or method.

Innermost Enclosing Exception Block: Given a statement S, *innermost-enclosing-exception-block(S)* is the innermost *exception block* containing S.

3 Overview of def-use algorithm

In [CRL99, CRL98] 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 like C++ and Java. The def-use algorithm presented in this paper is an application of relevant context inference to finding def-use associations; it uses the points-to algorithm presented in [CRL99, CRL98] and has the same overall structure as the points-to algorithm. First we will give a brief overview of the various steps of the def-use algorithm and 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; however, we have made every attempt to ensure that this paper can be read without reading [CRL99, CRL98].

The def-use algorithm (**Def-Use-Algo**) is an iterative worklist algorithm that is flow- and context-sensitive. Def-Use-Algo takes as input a statement-level interprocedural control flow graph (**ICFG**) [LR92]. From this, first an initial approximate call graph is formed and then decomposed into strongly connected components (SCC's). The following four phases are performed using the SCC condensation (**SCC-DAG**):

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].[4] 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 [BS96]); however, in practice we have found hierarchy analysis to be adequate.

Phase I: 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. It performs the following two subphases on each SCC.

Phase I-pta: During this 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**. The pointer summary transfer function of a method M is the set of data-flow elements (representing values of pointers) that reach the exit node of M and that do not represent values of local variables of M. This function summarizes the possible effects of method invocation on values of pointers. The 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 the 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 *a1* and *a2* are aliased at the entry of *method1*, and hence Phase 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 Phase 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, using the values of the unknown initial values of the method at the call site, to propagate data-flow elements from the exit of the method to the call site in a context-sensitive manner.

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

▶**Relevant contexts** Rather than calculate all possible conditions, the algorithm calculates only those conditions which may affect points-to 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 $a3_{init}$.

Phase I-dua: During this subphase Def-Use-Algo finds def-use associations using the points-to solution computed in Phase I-pta. For each method Def-Use-Algo computes, in terms of unknown initial values, 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**. The def-use summary transfer function of a method M is the set of data-flow elements (representing variable definitions) that reach the exit node of M and that do not represent definitions of local variables of M. This function summarizes the possible effects of method invocation on variable definitions. 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 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** Like *points-tos*, the def-use associations 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 association 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 association 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 association 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 $a1_{init} \in \{C\}$.

▶**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 of the method to the statement, the definition point of the corresponding def-use association 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. 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 of *method1*, and hence the definition point of the def-use association due to the definition and use of $a2_{init}.next$, computed at statement **6**, is parametrized by the unknown initial def of $a2_{init}.next$.

Phase II: Def-Use-Algo traverses the SCC-DAG in a topological order (top-down) and performs the following two

subphases on each SCC.

Phase II-pta: In this 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, Phase II-pta propagates $object_9$ from call site **11** to the entry of *method1* as a concrete value of $a1_{init}$. ▶It treats an interface initial value like a concrete value and propagates an **interface initial value** to the entry nodes of other methods if, through an actual-to-uiv 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, it propagates the interface initial value $a4_{init}$ to the entry of *method1* as a concrete value of $a1_{init}$.

Phase II-dua: In this 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, and hence at call site **11**, Phase II-dua propagates program point **10** to the entry of *method1* as a concrete value of the unknown initial def of $a2_{init}.next$. ▶Again, Phase II-dua treats an **interface initial def** like a concrete value and propagates an interface initial def to the entry nodes of other methods if, through a def-to-uidef 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** in Figure 1, it propagates the interface initial def $a5_{init}.next_{initdef}$ to the entry of *method1* as a concrete value of the unknown initial def of $a2_{init}.next$.

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.

This phase involves only the entry nodes and call nodes, and as the empirical results for points-to analysis in [CRL99] show, it is extremely fast.

Phase III: This phase involves only non-entry nodes. In this phase the unknown initial values and the unknown initial defs in the parametrized def-use solution computed during Phase I-dua are **instantiated** by their concrete values computed in Phase II. 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. This phase is completely *demand-driven*, and *needs to be performed only at those nodes where the final solution is needed*. After this phase, the def-use solution at each node is expressed entirely in terms of program variables, heap-names, interface initial values, definition points, interface initial defs and use points.

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 Phase I-pta) or def-use summary transfer function (during Phase 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 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.

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

2. There are five kinds of **DUA-dfelms**:
   DUA-dfelm-a. [*ECFInfo*, *relevant context-1*, ⟨*var, def-point,* ⟨*relevant context-2, value*⟩⟩]
   DUA-dfelm-b. [**excp-def,** *relevant context*, *exception object*, *throw-point*]
   DUA-dfelm-c. [*relevant context*, ⟨*var, def-point, use-point*⟩]
   DUA-dfelm-d. [*relevant context*, ⟨*exception-object, throw-point, catch-point*⟩]
   DUA-dfelm-e. [*relevant context*, ⟨*var, def-point, use-edge*⟩]

Figure 3: ▶Phase I data-flow elements

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] 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

4.1 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, are computed during Phase I-pta, and represent (a) values of pointer variables and (b) exception objects. *DUA-dfelms* are used for def-use analysis, are computed during Phase I-dua, and represent variable definitions and def-use associations.

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

1. Data-flow elements for points-to analysis: A **PTA-dfelm-a** represents a value of a pointer. An **ECFInfo** or *exception control-flow information* is one of the following:

- *excp-type*, the run-time type of an exception object,
- *iv*, when the unknown initial value *iv* is thrown as an exception object or
- *empty*, for propagation along a path from the entry node of a method to a statement in the method such that there is no uncaught exception along the path.

▶Intuitively, a *non-empty ECFInfo* stores the signature of an uncaught exception along a path through which a *PTA-dfelm* or *DUA-dfelm* reaches a program point, and determines the future propagation of the *PTA-dfelm* or *DUA-dfelm* from the program point. For example, statement **14** in Figure 1 throws $e1_{init}$ and hence when the *PTA-dfelm* representing the value of *global3* is propagated across statement **14**, $e1_{init}$ becomes the *ECFInfo* of the new *PTA-dfelm*-**a** propagated to statement **15**. The reason for storing an unknown initial value *uiv* as the signature of exception when *uiv* is thrown as the exception object is that the concrete type of the exception can be *typeof(uiv)* or any of its subtypes. 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 *PTA-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 **points-to** represents a pair of the form:

⟨*var, object*⟩.

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. *object* can also be *null*, which is treated as a special heap-name.

A **relevant context** has the form:

⟨*alias context, type context, ExcpObjTypeCont*⟩.

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 **eq** uiv_2*)*

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

(uiv_1 **neq** uiv_2*)*.

Here uiv_1 and uiv_2 are unknown initial values.

A **type context** is *empty* or it is a conjunction of type constraints. 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 *PTA-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*}. ▶A type constraint in a *type context* is inferred when an unknown initial value is the receiver of a dynamic dispatch.

ExcpObjTypeCont or *exception object type context* is *empty* or it is a conjunction of type constraints of 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*. ▶The type constraints in an *ExcpObjTypeCont* are inferred when an unknown initial value is propagated as an exception object.

▶These relevant contexts are inferred by the algorithm during 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 a 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.

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

A **PTA-dfelm-b** is used for propagating an exception object. *exception object* is either an unknown initial value or a heap-name. Intuitively, *PTA-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*.

2. Data-flow elements for def-use analysis:

DUA-dfelm-a. [*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. ▶*DUA-dfelms*-**a** are used for propagating variable definitions to their use points. For example, [*empty,*⟨($a1_{init}$ **eq** $a2_{init}$)*,empty,empty*⟩,⟨$a2_{init}$*.next,4,*⟨*Empty-context,*$a3_{init}$⟩⟩] represents the potential definition of $a2_{init}$*.next* at statement **4** in Figure 1.

DUA-dfelm-b. [**excp-def,** *relevant context*, *exception object*, *throw-point*], where (1) *throw-point* is the program point at which *exception object* has been thrown, (2) *exception object* is an unknown initial value or a heap-name and (3) **excp-def** is a key-word. *relevant context* represents the contexts in which *exception object* is thrown at *throw-point*. For example, [**excp-def,***Empty-context*,$e1_{init}$,*14*] represents the exception thrown at statement **14** in Figure 1. ▶*DUA-dfelms*-**b** are used for propagating exception definitions from throws to corresponding catches.

DUA-dfelm-c. [*relevant context*, ⟨*var, def-point, use-point*⟩], where (1) *var* and (2) *def-point* are same as in *DUA-dfelm*-**a**, and (3) *use-point* is a program point. *DUA-dfelms*-**c** capture def-uses of variables, although these are parametrized in terms of unknown initial defs and unknown initial values. ▶Intuitively, *var* is a variable 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 *DUA-dfelm*-**c 6:1** in Figure 11 shows the def-use of $a2_{init}$*.next* between statements **4** and **6** in Figure 1.

DUA-dfelm-d. [*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. ▶A *DUA-dfelm*-**d** represents a def-use association between a throw and a catch, it means *exception-object* thrown at *throw-point* in contexts represented by *relevant context* is caught at *catch-point*. For example, [⟨*empty,empty,(type(*$e1_{init}$*) ≤ ET2)*⟩,⟨$e1_{init}$*,14,16*⟩] represents the def-use between the throw at statement **14** and the catch at statement **16** in Figure 1.

DUA-dfelm-e. [*relevant context*, ⟨*var, def-point, use-edge*⟩], where (1) *use-edge* is an edge out of a condition node

```
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 4: Phase I

(representing the test expression of a *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 *DUA-dfelm*-**a**. ▶These *DUA-dfelms* represent *p-uses*. For example,
$[\langle empty, (type(a1_{init}) \in A::update), empty\rangle, \langle r,3,\langle 5,A::update\rangle\rangle]$
and
$[\langle empty, (type(a1_{init}) \in C::update), empty\rangle, \langle r,3,\langle 5,C::update\rangle\rangle]$
represent the *p-uses* due to dynamic dispatch at statement **5** in Figure 1.

Limiting relevant context 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]:

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.

4.2 Propagation of Data-flow Elements

This section presents the def-use-analysis transfer function for each kind of *ICFG* node. We will illustrate the different steps of the algorithm using the example in Figure 1. A high-level pseudocode for Phase I is given in Figure 4. 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( ) {
  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 5: initialize worklist for Phase I-dua

initialize-worklist-I-dua (Figure 5) 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 A; here we will explain them informally.

propagate-unknown-init-defs-from-entry-node (Figure 5) initializes the worklist with *DUA-dfelms*-**a** 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 5) initializes the worklist with *DUA-dfelms*-**a** representing defs generated at reachable assignment nodes, by using the points-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 $[empty, Empty\text{-}context, \langle a1_{init}.next,4,\langle Empty\text{-}context,a3_{init}\rangle\rangle]$, and this def implies the ▶potential def $[empty, \langle (a1_{init}\ \textbf{eq}\ a2_{init}), empty, empty\rangle, \langle a2_{init}.next,4,\langle Empty\text{-}context,a3_{init}\rangle\rangle]$, both of which are added to the worklist.

propagate-exception-objects-from-throw-node (Figure 5) initiates the propagation of *DUA-dfelms*-**b** 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, [**excp-def,***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.

```
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 6: process worklist for Phase I-dua

propagate-defs-created-at-an-object-creation-site(Figure 5) initializes the *worklist* with a *DUA-dfelm*-**a** 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 *DUA-dfelms*-**a** representing these definitions to the *worklist*.

▶**back-bind-using-def-use-summary-transfer-function (Figure 5)** 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*,$\langle a1_{init}.next,4,\langle$*Empty-context*,$a3_{init}\rangle\rangle$] reaches the exit of *method1* and hence this *DUA-dfelm*-**a** 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 *DUA-dfelm*-**a** is instantiated to [*empty*,*Empty-context*,$\langle a4_{init}.next,4,\langle$*Empty-context*,$a6_{init}\rangle\rangle$], which is added to the worklist.

process-worklist-I-dua(Figure 6) 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 7): Figure 7 defines the def-use-analysis transfer functions of assign-

```
/*** rdfe is DUA-dfelm-a [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 7: Transfer functions for assignment, condition, new and call nodes

ment, condition and new (object creation site) nodes. ▶*generate-dfelms-due-to-pot-non-aliases(n, rdfe)* (Appendix B) conditions the propagation of *rdfe* across n on potential non-aliases. For example, when *rdfe* is the *DUA-dfelm*-**a** [*empty*,*Empty-context*, $\langle a2_{init}.next, a2_{init}.next_{initdef}, \langle$*Empty-context*,$a2_{init}.next_{init}\rangle\rangle$] at statement **4** in Figure 1, *generate-dfelms-due-to-pot-non-aliases* generates the set of *DUA-dfelms*-**a**
{[*empty*,$\langle(a1_{init}$ *neq* $a2_{init})$,*empty*,*empty*$\rangle$, $\langle a2_{init}.next, a2_{init}.next_{initdef}, \langle$*Empty-context*,$a2_{init}.next_{init}\rangle\rangle$]}.

4. process-call (Figure 7): ▶**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*,$\langle r,3,\langle$*Empty-context*,$a1_{init}\rangle\rangle$] 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 *DUA-dfelms*-**e**
[$\langle$*empty*,(*type*($a1_{init}$)$\in$*A::update*),*empty*$\rangle$,$\langle r,3,\langle 5,$*A::update*$\rangle\rangle$] and [$\langle$*empty*,(*type*($a1_{init}$)$\in$*C::update*),*empty*$\rangle$,$\langle r,3,\langle 5,$*C::update*$\rangle\rangle$] are added to **5**.*def-uses*.

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*,$\langle$*var*,*def*$\rangle$,*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 5 and 6. *n.def-uidef-bindings* contains these bindings between the unknown initial defs and their values at n. For example in Figure 1, [*Empty-context*,$\langle object_{10}.next, 10\rangle$,$a2_{init}.next_{initdef}$] $\in$ **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 C.

If any of the targets invocable from n throws an exception that is not caught within the target, a new *DUA-dfelm*-**a** 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 8) 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 *DUA-dfelm*-**a**. The *ECFInfo* determines the future propagation of the generated *DUA-dfelm*-**a** 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*,$\langle$*global3,13,*$\langle$*Empty-context*,*object*$_{13}\rangle\rangle$] reaches statement **14** (from statement **13**), the *DUA-dfelm*-**a**
[$e1_{init}$,*Empty-context*,$\langle$*global3,13,*$\langle$*Empty-context*,*object*$_{13}\rangle\rangle$] is generated.

6. process-try-exit (Figure 8): 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-*

```
5. process-throw( n, rdfe ) {
/*** rdfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] ***/
new-DUA-dfelms-a = 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-DUA-dfelms-a = new-DUA-dfelms-a ∪
                        ▶{[typeof(eo),rc4,⟨var,def,⟨rc2,val⟩⟩]}
    else
      /*** eo is an unknown initial value ***/
      new-DUA-dfelms-a = new-DUA-dfelms-a ∪
                        ▶{[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-DUA-dfelms-a)
}
```

```
6. PROCESS-TRY-EXIT( n, rdfe ) {
/*** first case:rdfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
      or
second case:
      rdfe is DUA-dfelm-b [excp-def,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 = [excp-def,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 8: Transfer functions for throw and try-exit

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 *DUA-dfelm*-**a** [$e1_{init}$,*Empty-context*,⟨*global3,13,*⟨*Empty-context,object*$_{13}$⟩⟩] at program point **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 *DUA-dfelm*-**a** [$e1_{init}$,⟨*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 *DUA-dfelm*-**b [excp-def,***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 *DUA-dfelm*-**b [excp-def,**⟨*empty,empty,(type*(e_{init}) $\not\leq$ *ET2)*⟩,$e1_{init}$,*14*] is propagated to program point 18.

7. process-catch (Figure 9): 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 association 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 *DUA-dfelm*-**a** 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: *DUA-dfelm*-**a** [$e1_{init}$,*Empty-context*,⟨*global3,13,*⟨*Empty-context,object*$_{13}$⟩⟩] and *DUA-dfelm*-**b [excp-def,***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 *DUA-dfelm*-**a** [*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 *DUA-dfelm*-**d** [⟨*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 9): 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 D 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 D defines def-use-analysis transfer function for return statements.

5 Phase II

A high-level description of Phase II is given in Figure 10.

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, 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. During Phase II-pta, Def-Use-Algo treats an interface initial value like a concrete value and propagates an interface initial value to the entry nodes of other methods if through an actual-to-uiv binding at a call site the interface initial value is the value of an unknown initial value at a target entry.

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

initialize-worklist-II-dua (Figure 10) 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 a non-interface-initial-def unknown 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: 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*,⟨$a5_{init}$.*next*,$a5_{init}$.$next_{initdef}$⟩,$a2_{init}$.$next_{initdef}$] ∈ **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 10) 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.

```
/*** first case:  rdfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
            or
second case:
      rdfe is DUA-dfelm-b [excp-def,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 9: Transfer functions for catch entry and catch exit nodes

```
for each SCC in topological order
  phaseII-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 10: phase II

6 Phase III

In this phase unknown initial values and unknown initial defs in *DUA-dfelms*-**c**, *DUA-dfelms*-**d** and *DUA-dfelms*-**e** 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 Phase II-pta; unknown initial defs are instantiated by their concrete values computed in Phase II-dua. Whenever a $\langle obj,\ def\rangle$ is used to instantiate an unknown initial def *uidef1* in a *DUA-dfelm*-**c** [*rc,* $\langle$*var1, uidef1, use-point*$\rangle$] or a *DUA-dfelm*-**e** [*rc,* $\langle$*var1, uidef1, use-edge*$\rangle$], *uidef1* is replaced by *def* and *var1* is replaced by *obj*. This pairing of *obj* and *def* increases precision by not mixing *def*'s and *obj*'s reaching the corresponding entry node from different paths. 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 the instantiated relevant contexts can evaluate to true by making conservative, worst-case assumptions about these interface initial values, yield the final def-use solution. Note that a potential non-alias 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. Each element of the final def-use solution at a node n has one of the following three forms:

1. [*rc,* $\langle$*var, def-point, use-point*$\rangle$], where *def-point* is a program point or an interface initial def, *use-point* is n and *var* is a global, local, field of a heap-name or field of an interface initial value. This is used for def-uses of variables (including flow due to exceptions).

2. [*rc,* $\langle$*excp-object, throw-point, catch-point*$\rangle$], where *throw-point* is a throw statement number, *catch-point* is n and *excp-object* is a heap-name or an interface initial value. This is used for def-uses between throws and catches.

3. [*rc,* $\langle$*var, def-point, use-edge*$\rangle$], 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.

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* that 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 *DUA-dfelm*-**c** [$\langle$*empty,(type(*$a1_{init}$*)∈A::update),empty*$\rangle$,$\langle$*global1,1,7*$\rangle$] is instantiated to [$\langle$*empty,(type(*$a4_{init}$*)∈A::update),empty*$\rangle$,$\langle$*global1,1,7*$\rangle$] because $a4_{init}$ is an interface initial value and it a possible concrete value of $a1_{init}$ 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 associations 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 association (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 association. If only the def-use solution is required then the relevant contexts can be dropped from the final solution.

7 Algorithm for finding reduced context cover

In this section we discuss some ways in which the relevant contexts of def-use associations, involving only interface initial values of a single *public interface method*, computed during Phase III, can be used in designing relevant test cases for satisfying testing criteria based on def-use associations.

▶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 associations computed by Phase III (see Section 6) for a library **L**. Let $\mathbf{RelevantContextCover_M}$ be the subset of $\mathbf{RelevantContextCover_L}$ such that each element of $RelevantContextCover_M$ involves only 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 to M. ▶Alternatively, the tester may choose to specify the values of some of the globals or parameters of pointer type partially, and thus provide a test case template from which many test cases can be generated. In the second case, at the call to M, for each global or parameter v of pointer type that the tester wants to specify only partially, the tester may do one of the following: (1) indicate that v is completely unspecified or (2) specify the value of v using interface initial values. The value of each pointer field of an interface initial value used in the specification 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 value of each non-pointer field of an interface initial value used in the specification needs to be specified completely because in this paper we keep track of only definitions of non-pointer variables and not their values. For each interface initial value, the tester may optionally choose to specify its concrete type also.

Given a template **t** for a call to M, let $\mathbf{ConsistentRelContCover_{M_t}}$

be the subset of $RelevantContextCover_M$ such that each element of $ConsistentRelContCover_{M_t}$ is consistent with the specification of t. One strategy for the tester is to manually[5] enhance t using each element of $ConsistentRelContCover_{M_t}$ and generate *size-of($ConsistentRelContCover_{M_t}$)* test cases (one test case per element of $ConsistentRelContCover_{M_t}$) which can be executed separately. ▶However, there is a more efficient alternative: first the tester can compute a reduced context cover **ReducedCover** of $ConsistentRelContCover_{M_t}$ having as small a size as possible and then generate *size-of(ReducedCover)* test cases by enhancing t using the elements of *ReducedCover*. Intuitively, the goal is to cover as many def-use associations 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.

For any pointer-type global, parameter or field of an interface initial value whose value is not mentioned in the template specification or in the relevant context used to enhance the template, the following can be done: If the variable or field has been read, any object of the declared type of the variable or field can be assigned to it. Otherwise, it can be assigned *null* as its initial value is not relevant for data-flow testing. For non-pointer-type fields of objects introduced during enhancement, one of the following alternatives can be used: (1) the tester may assign some values or (2) default values for the types of these fields may be assigned.

Given a template t of a call to a *public interface method* M and $ConsistentRelContCover_{M_t}$, the following simple greedy algorithm can be used for finding a reduced context cover *ReducedCover* of $ConsistentRelContCover_{M_t}$ such that each element of *ReducedCover* is consistent with t, although it may not find the smallest sized reduced context cover in some cases:

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

For example, $RelevantContextCover_{L_{example}}$ for $L_{example}$ given in Figure 1 is
{ $\langle empty \rangle$, $\langle (a4_{init}$ **eq** $a5_{init}) \rangle$, $\langle (a4_{init}$ **neq** $a5_{init}) \rangle$, $\langle (type(a4_{init}) \in A::update) \rangle$, $\langle (type(a4_{init}) \in C::update) \rangle$, $\langle (type(e2_{init}) \leq ET2) \rangle$, $\langle (type(e2_{init}) \leq ET4) \rangle$, $\langle (type(e2_{init}) \not\leq ET2) \wedge (type(e2_{init}) \not\leq ET4) \rangle$}. The relevant def-uses whose relevant contexts are the non-empty elements of $RelevantContextCover_{L_{example}}$ are shown in Figure 12. Figure 11 shows the def-uses computed by Phase I and Figure 12 shows the same def-uses after the unknown initial values and unknown initial defs have been instantiated by their concrete values in Phase III.

Now consider a template *t1* of a call to the *public interface method method2* of $L_{example}$ where all pointer variables are left unspecified by the tester. $RelevantContextCover_{method2}$ is
{ $\langle empty \rangle$, $\langle (a4_{init}$ **eq** $a5_{init}) \rangle$, $\langle (a4_{init}$ **neq** $a5_{init}) \rangle$, $\langle (type(a4_{init}) \in A::update) \rangle$, $\langle (type(a4_{init}) \in C::update) \rangle$} and $ConsistentRelContCover_{method2_{t1}}$ is same as $RelevantContextCover_{method2}$. As a result, the reduced context cover *ReducedCover* computed by the algorithm given above is { $\langle (a4_{init}$ **eq** $a5_{init}) \wedge (type(a4_{init}) \in A::update) \rangle$, $\langle (a4_{init}$ **neq** $a5_{init}) \wedge (type(a4_{init}) \in C::update) \rangle$}.

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

8 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 [CRL98, CRL97]. 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-uiv 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 *ExcpObjTypeCont*'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 *ExcpObjTypeCont*'s provide a measure of code complexity. Complicated alias contexts, type contexts or *ExcpObjTypeCont*'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

6:1 [$\langle (a1_{init}$ **eq** $a2_{init}), empty, empty\rangle$,$\langle a2_{init}.next, 4, 6\rangle$]
6:2 [$\langle (a1_{init}$ **neq** $a2_{init}), empty, empty\rangle$,$\langle a2_{init}.next, a2_{init}.next_{initdef}, 6\rangle$]

7:1 [$\langle empty, (type(a1_{init}) \in A::update), empty\rangle$,$\langle global1, 1, 7\rangle$]
7:2 [$\langle empty, (type(a1_{init}) \in C::update), empty\rangle$,$\langle global1, 2, 7\rangle$]

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

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

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

Figure 11: Part of Phase I def-use solution (DUA-*dfelms*-**c**)

`/*** 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 [$\langle (a4_{init}$ **eq** $a5_{init}), empty, empty\rangle$,$\langle a5_{init}.next, 4, 6\rangle$]
6:2 [$\langle (a4_{init}$ **neq** $a5_{init}), empty, empty\rangle$,$\langle a5_{init}.next, a5_{init}.next_{initdef}, 6\rangle$]

7:1 [$\langle empty, (type(a4_{init}) \in A::update), empty\rangle$,$\langle global1, 1, 7\rangle$]
7:2 [$\langle empty, (type(a4_{init}) \in C::update), empty\rangle$,$\langle global1, 2, 7\rangle$]

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

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

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

Figure 12: Part of Phase III def-use solution

instrumented to observe def-use associations executed by a set of test cases.

9 Related Work

Data-flow-based testing has a long history [FO76, Ost77, LK83, RW85, FW88, LCS89, HS89, OW91]. [PLR94] presents a def-use analysis algorithm for C programs. [HR94] applied the algorithm in [PLR94] to C++ classes. As stated earlier, the approach presented in [HR94] cannot compute the potential def-uses due to possible aliasing at the entry node of a *public interface method*. [PLR94, HR94] did not consider flow due to exceptions. Another advantage of our work is that it can be easily incorporated with the approach presented in [HR94]. [HS94] presents an algorithm for computing def-uses in Fortran programs. [HFGO94] presents an empirical investigation of finding errors using data-flow-based testing criteria. [RBS97] presents an algorithm for refining def-use solution using information about infeasible paths.

There are many tools [PFW85, Ost90] for checking def-use coverage.

10 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.

References

[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.

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

[CRL97] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Complexity of concrete type-inference in the presence of exceptions. Technical Report DCS-TR-341, Dept of CS, Rutgers University, September 1997.

[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.

[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.

[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.

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

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, May 1994.

[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.

[HS89] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[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.

[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.

[Ost77] L. J. Osterweil. The detection of unexecutable program paths through static data-flow analysis. In *In Proceedings of IEEE COMPSAC 77*, pages 406–413, December 1977.

[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.

[RBS97] Rajiv Gupta Rastislav Bodik and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Proceedings of the Fifth Symposium on the Foundations of Software Engineering*, September 1997.

[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.

A Auxiliary functions of init-worklist-I-dua

```
1. propagate-unknown-init-defs-from-entry-node( n ) {
/*** it propagates DUA-dfelms-a ***/
  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-DUA-dfelms-a = 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-DUA-dfelms-a = gen-DUA-dfelms-a ∪ {y}
      gen-DUA-dfelms-a = gen-DUA-dfelms-a ∪
        ▶generate-defs-due-to-pot-aliases(n.method, y)

  add-to-soln-and-worklist-if-needed-I(n.successor, gen-DUA-dfelms-a)
}
▶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-DUA-dfelms-b = φ
  for each exception object eo thrown by n
    let rc3 be the relevant context under which eo
          is thrown by n
    gen-DUA-dfelms-b = gen-DUA-dfelms-b ∪
                       {[excp-def,rc3,eo,n]}
    successor = exit node of innermost-enclosing-exception-block(n)
  add-to-soln-and-worklist-if-needed-I(successor, gen-DUA-dfelms-b)
}
```

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

```
5. back-bind-using-def-use-summary-transfer-function(m, c) {
  for each dfe ∈ m.exit-node.reaching-defs
    /*** dfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩] or
    dfe is DUA-dfelm-b [excp-def,rc1,excp-obj,throw-point] ****/
    if ((dfe is a DUA-dfelm-b) 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 [excp-def,rc3,excp-obj1,throw-point]'s
      to which dfe maps at c ***/
      z = x
      for each DUA-dfelm-a 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 *DUA-dfelms*-**a** 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 **{**⟨⟨(*a*1$_{init}$ **eq** *a*2$_{init}$*),empty,empty*⟩,*a*3$_{init}$⟩, ⟨⟨(*a*1$_{init}$ **neq** *a*2$_{init}$*),empty,empty*⟩,*a*2$_{init}$*.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 *PTA-dfelms-***a** [*empty,*⟨(*a*1$_{init}$ **eq** *a*2$_{init}$*),empty,empty*⟩,⟨*a*2$_{init}$*.next,a*3$_{init}$⟩] and [*empty,*⟨(*a*1$_{init}$ **neq** *a*2$_{init}$*),empty,empty*⟩,⟨*a*2$_{init}$*.next,a*2$_{init}$*.next*$_{init}$⟩]**}** on the top of statement **6**.

▶**generate-defs-due-to-pot-aliases** generates *DUA-dfelms-***a** 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 *DUA-dfelms-***a** [*empty*, ⟨(*a*1$_{init}$ *eq* *a*2$_{init}$*),empty,empty*⟩, ⟨*a*2$_{init}$*.next,4,*⟨*Empty-context,* *a*3$_{init}$⟩⟩] when *method* is *method2* and *dfe* is [*empty*, *Empty-context*, ⟨*a*1$_{init}$*.next,4,*⟨*Empty-context,* *a*3$_{init}$⟩⟩].

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.

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

```
▶1.a generate-dfelms-due-to-pot-non-aliases(n, rdfe) {
/*** rdfe is DUA-dfelm-a [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
```

C 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 DUA-dfelm-e 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

{ $[\langle empty, (type(a1_{init}) \in A::update), empty\rangle, global1_{init}, A::update.global1_{init}]$,
$[\langle empty, (type(a1_{init}) \in C::update), empty\rangle, global1_{init}, C::update.global1_{init}]$,
$[\langle empty, (type(a1_{init}) \in A::update), empty\rangle, a1_{init}, A::update.this_{init}]$,
$[\langle empty, (type(a1_{init}) \in C::update), empty\rangle, a1_{init}, C::update.this_{init}]$}

```
/*** first case:  rdfe is DUA-dfelm-a [ecfi,rc1,⟨var,def,⟨rc2,val⟩⟩]
                                                                  ]or
     second case:
       rdfe is DUA-dfelm-b [excp-def,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 [excp-def,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

D 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 DUA-dfelm-a [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