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 Lexample 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 Lexample, 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-
ship between statements 13 and 17. For def-use coverage, such def-uses between throw and catch, 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. It includes single inheritance, dynamic dispatch, recursive types, exceptions and pointer assignment statements with a single level of dereferencing. Any pointer assignment statement with any levels of dereferencing can be translated to this form using temporaries. The semantics of the constructs is same as in C++. All exceptions include single inheritance, dynamic dispatch, recursive types, exceptions and pointer assignment statements with a single level of dereferencing. Any pointer assignment statement with any levels of dereferencing can be translated to this form using temporaries. (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 with the single name object_n, a heap-name.

Unknown initial value(UIV): We use varinit 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,

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

In the class definitions, the terminal symbol function signatures are underlined. (any pointer assignment statement with any levels of dereferencing can be translated to this form using temporaries).

![Figure 1: Library L_example](image-url)
Program ⇒ \{Class | Proc\}+

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

DataMember ⇒ Protection \{static \} Type FieldName \\
Type ⇒ ClassName \| PreimitiveType

PreimitiveType ⇒ 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 Stmt\+

Decls ⇒ Decl*

Decl ⇒ Type VarName \\
Stmt ⇒ AssignmentStmt \| NewStmt \| Call \| If \| While\| ReturnStmt \| Try \| Throw \| i

ReturnStmt ⇒ return VarName \\
AssignmentStmt ⇒ Lhs \= Rhs:

Lhs ⇒ VarName \| VarName.FieldName

Rhs ⇒ VarName \| VarName.FieldName \| 0 \| Expr

Call ⇒ \{VarName = \} \{VarName.MethodName\MethodName | ClassName.MethodName\|ProcName\} \{ [VarName RestVar\} \} \\
NewStmt ⇒ VarName = new ClassName \{ [VarName RestVar\} \} \\
Stmt ⇒ AssignmentStmt \| NewStmt \| Call \| If \| While\| ReturnStmt \| Try \| Throw \| i

ReturnVar ⇒ VarName

If ⇒ if \{ Expr \} Stmt\+ \{ else \{ Stmt\+ \} \}

While ⇒ while \{ Expr \} Stmt\+

Throw ⇒ throw VarName

Try ⇒ try \{ Stmt\+ \} Catch*

Catch ⇒ catch \{ ClassName \= VarName \} \{ Stmt\+ \}

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→next points and var_{init}, next_{init}, next_{init}\) represents the unknown initial value to which var→next→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 \(var_{init,def}\) 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 \(use-point\) 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]. 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 [CLR99, 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 dependencies, so they are computed simultaneously by fixed point iteration. In contrast, the pointer summary transfer functions of methods in different SCC’s have hierarchical 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 of these unknown initial values. For example, statement 4 in Figure 1 can modify the next field of a2init if a1 and a2 are aliased at the entry of method1, and hence Phase I-pta infers that on the top of statement 5, a2next points to a3init under the condition that a1init and a2init are the same object at the entry of method1. Similarly, statement 5 can invoke A::update or C::update depending upon the concrete type of a1init, and hence Phase I-pta infers that on the top of statement 6, global1 points to object1 under the condition that the concrete type of a1init is \{A, B\} and global1 points to object2 under the condition that the concrete type of a1init is \{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.

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 a1init and a2init, but it does not use the next field of a3init.

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-to's, 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 a2next at statement 4 and later its use at statement 6 is computed dependent on the condition that a1init and a2init are the same object at the entry of method1. Similarly, the def-use 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 a1init is \{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 a1init is \{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 a2next 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 a2next is used at statement 6 is parametrized by the unknown initial def of a2next.

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

---

1 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.
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 object1 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-uv 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 object10 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-uidf 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 to the entry of method1 as a concrete value of the unknown initial def of a2_{init}.

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, heapnames, 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-dfels:
   PTA-dfelm-a. [ECFInfo, relevant context, points-to]
   PTA-dfelm-b. [relevant context, exception object]

2. There are five kinds of DUA-dfels:
   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 type.

4 Phase I

4.1 Data-flow Elements

Def-Use-Algo computes two kinds of data-flow elements during phase I: PTA-dfels and DUA-dfels. These are shown in Figure 3. PTA-dfels are used for points-to analysis, are computed during Phase I-pta, and represent (a) values of pointer variables and (b) exception objects. DUA-dfels 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 uv as the signature of an exception when uv is thrown as the exception object is that the concrete type of the exception can be type(foo) 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 \( m3 \) 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:

\[
(\text{alias context}, \text{type context}, \text{ExcpObjTypeCont})
\]

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:

\[
(uiv1 \ eq \ uiv2)
\]

and each potential non-alias has the form:

\[
(uiv1 \ neq \ uiv2)
\]

Here \( uiv1 \) and \( uiv2 \) 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 \( dfelm \) hold. When a \( dfelm \) \( dfc \) 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 \( dfc \): the conjunct evaluates to true, it evaluates to false (\( dfc \) 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-dfelm-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:**

**DUAl-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-dfelm. **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.

**DUAl-dfelm-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.

**DUAl-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, \( [excp-def,Empty-context,e1_{init},4] \) represents the exception thrown at statement 14 in Figure 1.

**DUAl-dfelm-b** are used for propagating exception definitions from throws to corresponding catches.

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

**DUAl-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. **DUAl-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, \( [(empty,empty,(type(e1_{init}) \leq ET2)),(e1_{init},14,16)] \) represents the def-use between the throw at statement 14 and the catch at statement 16 in Figure 1.

**DUAl-dfelm-e.** **[relevant context, (var, def-point, use-edge)]**, where (1) **use-edge** is an edge out of a condition node
This section presents the def-use-analysis transfer function.

4.2 Propagation of Data-flow Elements

The conjuncts are dropped.

For example, \( [\text{empty}, \text{type}(a_{\text{init}}) \in A::update], \text{empty} ] \), \( (r, 3, (5, A::update)) \) and \( [\text{empty}, \text{type}(a_{\text{init}}) \in C::update], \text{empty} ] \), \( (r, 3, (5, C::update)) \) 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.** Any dfelm with relevant context \( r \), 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 an example, each SCC contains exactly one method and \{A::update, C::update, method1, method2, method3, method4\} is a reverse topological ordering of the SCCs. 

**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 since 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** iteratively analyzes the current SCC to mark all the non-pointer fields of unknown initial values that are defined or read. This is needed because non-pointer fields are not considered in Phase I-pta. Iteration is needed because due to actual-to-actual bindings, the use of a field in a callee may imply the use of another field in the caller. Note that Phase I-pta has already marked the pointer fields of unknown initial values that have been found to be used. For example, in method1 in Figure 1, Phase I-pta marks a1_{init}.next as defined and a2_{init}.next as read. Since a3_{init}.next is not used in method1 or in any method invoked during the lifetime of method1, a3_{init}.next is not marked.

**initialize-worklist-I-dua** (Figure 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 \([empty, Empty-context, a1_{init}.next, 4, (Empty-context, a3_{init}.next)]\), and this def implies the potential def \([empty, a1_{init}.eq a2_{init}.next, empty, empty, a3_{init}.next, 4, (Empty-context, a3_{init}.next)]\), 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, \([\text{except}, Empty-context, e1_{init}.next, 14]\) is propagated to the exit node of innermost-enclosing-exception-block(14), i.e., program point 15, the exit node of the try statement.

---

**Figure 4:** Phase I

**Figure 5:** initialize worklist for Phase I-dua
process-worklist-I-dua( ) {
    while worklist is not empty {
        wl-node = delete from worklist
        *** if the worklist is empty, this function returns to the calling function
        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)
            process-condition( n, rdfe )
            *** i.e., n represents the test expression of *** if or while ***/
        3: if (n is an object creation node)
            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)
            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-dfelm-a representing these definitions to the worklist.

*back-bind-using-def-use-summary-transfer-function(Figure 5) instantiates the DUA-dfelm-a at the exit node of a non-same-SCC method m using actual-to-uv 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-uv bindings are computed by phase-I-pta. For example, [empty, Empty-context(a1.next, 4), Empty-context(a3.next, 4)] reaches the exit of methodI and hence this DUA-dfelm-a is part of the def-use summary transfer function of methodI. Thus, when Phase I-dua is done on method2, at call site 8, the above DUA-dfelm-a is instantiated to [empty, Empty-context(a1.next, 4), Empty-context(a3.next, 4)], 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-
ment, condition and new object creation site) nodes. \(\text{generate-dfelms-due-to-potential-aliases}(n, \text{rdfe})\) (Appendix B) conditions the propagation of \(\text{rdfe}\) across \(n\) on potential non-aliases. For example, when \(\text{rdfe}\) is the \(\text{DUA-dfelm-a}[\emptyset,\emptyset,\langle a_{\text{init}},\text{next},a_{\text{init}},\text{next}\rangle,\langle \text{Empty-context},a_{\text{init}},\text{next}\rangle]\) at statement 4 in Figure 1, \(\text{generate-dfelms-due-to-potential-aliases}\) generates the set of \(\text{DUA-dfelms-a}\) \{\langle \emptyset,\langle a_{\text{init}},\text{next}\rangle,\emptyset,\emptyset\rangle,\langle a_{\text{init}},\text{next},a_{\text{init}},\text{next}\rangle,\langle \text{Empty-context},a_{\text{init}},\text{next}\rangle\}\}. 4. process-call (Figure 7) \(\text{a. compute-p-uses-due-to-dynamic-dispatch}\) uses \(\text{val}\) to compute the \(\text{p-uses}\) due to dynamic dispatch mentioned in Section 1. For example, when \(\text{rdfe}\) is \[\langle \text{Empty-context},a_{\text{init}}\rangle\] at statement 5 in Figure 1, \(\text{val}\) is \(a_{\text{init}}\) and the possible targets with \(a_{\text{init}}\) as receiver are \(A: \text{update}\) and \(C: \text{update}\). As a result, the \(\text{DUA-dfelms-e}\) \[\langle \emptyset,\langle \text{type}(a_{\text{init}})\rangle,\emptyset,\langle r,3,\langle 5,\text{A: update}\rangle\rangle\] and \[\langle \emptyset,\langle \text{type}(a_{\text{init}})\rangle,\langle r,3,\langle 5,\text{C: update}\rangle\rangle\] are added to \(5. \text{def-uses}\). 4b. \(\text{compute-new-bindings-of-defs-and-uidefs}\) 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 \(\langle r_{\text{c}},\text{var,def},uidef\rangle\); here \(r_{\text{c}}\) is a relevant context under which the binding holds, \(\text{uidef}\) is an unknown initial def at the entry of a target of \(n\), \(\text{var}\) is a location whose reaching definitions are represented by \(\text{uidef}\) at the entry of the target, and \(\text{def}\) is a definition point or an unknown initial def that represents a definition of \(\text{var}\). The reason why \(\text{var}\) 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. \(\text{n.def-uidef-bindings}\) contains these bindings between the unknown initial defs and their values at \(n\). For example in Figure 1, \(\langle \text{Empty-context},\langle \text{object}_{\text{y}},\text{next},106,\langle a_{\text{init}},\text{next}\rangle\text{init}\rangle\) in \(11. \text{def-uidef-bindings}\). Details of compute-p-uses-due-to-dynamic-dispatch and compute-new-bindings-of-defs-and-uidefs 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 \(\text{DUA-dfelm-a}\) is computed whose \(\text{ECFInfo}\) stores the signature of the exception thrown by the target. If \(\text{excp-object}\) is a heap-name, get-ecfi(\(\text{excp-object}\)) returns \(\text{typeof(\text{excp-object})}\). Otherwise \(\text{excp-object}\) is an unknown initial value and get-ecfi(\(\text{excp-object}\)) returns \(\text{excp-object}\). 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 \(\text{ECFInfo}\) of the generated \(\text{DUA-dfelm-a}\). The \(\text{ECFInfo}\) determines the future propagation of the generated \(\text{DUA-dfelm-a}\) from the exit node of \(\text{innermost-enclosing-exception-block}(n)\). For example, at statement 14 in Figure 1, there is only one \(eo, e_{\text{init}}\), and the corresponding \(rc_{3}\) is \(\text{Empty-context}\). As a result, when the \(\text{rdfe}\) \[\langle \text{Empty-context},\langle \text{global}\rangle,13,\langle \text{Empty-context},\text{object}_{13}\rangle\rangle\] reaches statement 14 (from statement 13), the \(\text{DUA-dfelm-a}\) \[\langle e_{\text{init}},\text{Empty-context},\langle \text{global}\rangle,13,\langle \text{Empty-context},\text{object}_{13}\rangle\rangle\] is generated. 6. process-try-exit (Figure 8): There are two cases: either (1) \(\text{rdfe}\) represents a variable definition or (2) \(\text{rdfe}\) represents an exception object. If \(\text{rdfe}\) represents a variable definition and it flows to \(n\) along a path without any uncaught exception (i.e., \(e_{\text{c}}\) is empty), process-try-exit. **/ commenting functions for throw and try-exit

6. PROCESS-TRY-EXIT\((n, \text{rdfe})\) \{ /**/ first case: \(\text{rdfe}\) is \(\text{DUA-dfelm-a}[\text{ecfi,rc1,}\langle \text{var,def},(\text{rc2,va})\rangle]\) or second case: \(\text{rdfe}\) is \(\text{DUA-dfelm-b}[\text{exp-d,rc1,exp-obj,throw-point}]\) /**/ successors contains the nodes to which \(\text{rdfe}\) needs to be propagated /**/ successors = \(\emptyset\) if \(\text{rdfe}\) is \(\langle \text{empty},r,1,\langle \text{var,def},(\text{rc2,va})\rangle\rangle\) /**/ NORMAL TERMINATION /**/ successors = successors \(\cup\) \{ordinary successor of \(n\)\} else \{ /**/ TERMINATION DUE TO EXCEPTION /**/ either first case with non-empty \(\text{ecfi}\) or second case /**/ /**/ FIND APPROPRIATE CATCHES /**/ for each catch \(c\) associated with \(n\) that can catch the exception represented by \(\text{ecfi}\) or \(\text{exp-obj}\) successors = successors \(\cup\) \{ \(c\).entry\} /**/ Can the exception ESCAPE all the catches? /**/ if (the exception represented by \(\text{ecfi}\) or \(\text{exp-obj}\) can escape all the catches associated with \(n\)) a: propagate-escaped-exception\((n, \text{rdfe})\) for each succ in successors add-to-soln-and-worklist-if-needed-I\((\text{succ},\text{new-DUA-dfelm-a})\) for each succ in successors add-to-soln-and-worklist-if-needed-I\((\text{succ},\text{new-DUA-dfelm-a})\) \} **/ 6a. propagate-escaped-exception\((n, \text{rdfe})\) \} let \(rc_{1}\) be \(\langle a_{\text{c}},tc_{\text{c}},\text{eotc}\rangle\) if (first case: \(\text{rdfe}\) represents a variable def) \(\text{new-rc}_{\text{1}} = (a_{\text{c}},tc_{\text{c}},\text{eotc}) \cup \{\text{get-escape-ecs}(n, e_{\text{c}})\}\) \(\text{new-DUA-dfelm} = [\langle \text{ecfi,new-rc}_{\text{1}},\langle \text{var,def},(\text{rc2,va})\rangle\rangle]\) else \{ /**/ second case: \(\text{rdfe}\) represents an exception object /**/ \(\text{new-rc}_{\text{2}} = (a_{\text{c}},tc_{\text{c}},\text{eotc}) \cup \{\text{get-escape-ecs}(n, \text{exp-obj})\}\) \(\text{new-DUA-dfelm} = [\langle \text{exp-d,new-rc}_{\text{2}},\text{exp-obj,throw-point}\rangle]\) \} \} **/ Figure 8: Transfer functions for throw and try-exit
exit propagates \( \text{rdfe} \) to the successor of \( n \), i.e., the statement following the try-catch construct. Otherwise, it propagates \( \text{rdfe} \) to the entry nodes of the catch statements that are associated with the try statement and that can catch the exception represented by \( \text{ecfi} \) or \( \text{excp-obj} \). If the exception can escape all of the catch statements associated with the try statement, process-try-exit propagates a new \( \text{DUA-dfelm}\)-a 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, \((\text{first case}) \) let \( \text{rdfe} \) be the \( \text{DUA-dfelm}\)-a \([e_{1,init},\text{Empty-context},(\text{global}3,13,\text{Empty-context},\text{object}_{13})]\) at program point 15 in Figure 1, the exit node of the try statement in method3. Since \( \text{typeof}(e_{1,init}) \) is compatible with ET2, successors is \([16]\) and \( \text{rdfe} \) is propagated to the entry node of the catch at program point 16. Moreover, the exception \((e_{1,init})\) 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 \((\text{type}(e_{1,init}) \not\leq \text{ET2})\) and the DUA-dfelm-a \([e_{1,init},\text{empty},\text{empty},(\text{type}(e_{1,init}) \not\leq \text{ET2});(\text{global}3,13,\text{Empty-context},\text{object}_{13})]\) is propagated to program point 18, i.e., exit node of innermost-enclosing-exception-block(15). Next, \((\text{second case}) \) let \( \text{rdfe} \) be the DUA-dfelm-b \([\text{excp-def},\text{Empty-context},e_{1,init},14]\) at program point 15. Due to the same reason as in the first case, \( \text{rdfe} \) is propagated to program point 16 and the DUA-dfelm-b \([\text{excp-def},\text{empty},\text{empty},(\text{type}(e_{1,init}) \not\leq \text{ET2});e_{1,init},14]\) is propagated to program point 18.

7. process-catch (Figure 9): If \( \text{rdfe} \) represents a variable definition, process-catch catches the exception represented by \( \text{ECFInfo} \) of \( \text{rdfe} \). If \( \text{rdfe} \) 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 \( \text{ECFInfo} \) or \( \text{excp-obj} \) 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 \( \text{rdfe} \)'s at this program point: DUA-dfelm-a \([e_{1,init},\text{Empty-context},(\text{global}3,13,\text{Empty-context},\text{object}_{13})]\) and DUA-dfelm-b \([\text{excp-def},\text{Empty-context},e_{1,init},14]\). In the first case, \( \text{ECFInfo} \) is \( e_{1,init} \) and hence get-catch-tcs returns \((\text{type}(e_{1,init}) \leq \text{ET2})\). As a result, new-dfe is the DUA-dfelm-a \([\text{empty},\text{empty},\text{empty},(\text{type}(e_{1,init}) \leq \text{ET2});(\text{global}3,13,\text{Empty-context},\text{object}_{13})]\). In the second case, \( \text{excp-obj} \) is \( e_{1,init} \) and again get-catch-tcs returns \((\text{type}(e_{1,init}) \leq \text{ET2})\). As a result, the DUA-dfelm-d \([(\text{empty},\text{empty},(\text{type}(e_{1,init}) \leq \text{ET2});(e_{1,init},14,16)]\) is added to the solution at program point 16 and \( \text{new-dfe} \equiv (\text{empty},\text{empty},(\text{type}(e_{1,init}) \leq \text{ET2}))\). \( \text{excp-16,Empty-context,e}_{1,init}\) is generated to represent a definition of the parameter of the catch statement.

8. process-catch-exit (Figure 9): If \( \text{rdfe} \) flows to \( n \) along a path without any uncaught exception (i.e., \( \text{ECFInfo} \) is empty), process-catch-exit propagates \( \text{rdfe} \) to the successor of the try-catch construct. Otherwise, process-catch-exit propagates \( \text{rdfe} \) to the exit node of innermost-enclosing-exception-block(\( n \)).

9.10. Return states 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-p$a$ta is same as the Phase II described in [CRL99, CRL98]. For this paper it is enough to know that Phase II-p$a$ta 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-p$a$ta. 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-p$a$ta, it considers only those methods in the current SCC that have been marked reachable. Whenever Def-Use-Algo finds an unmatched method to be invocable from a call site in a reachable method, it marks the new method also as reachable. During Phase II-p$a$ta, 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-ual binding at a call site the interface initial value is the value of an unknown initial value at a target entry.

initialize-worklist-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 def's 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-undef-binding instantiates the unknown initial values in the rc and var of a def-undef binding with their concrete values computed at the entry node of the corresponding method during Phase II-p$a$ta. It returns those instantiations for which \( \text{rc} \) can evaluate to true: either (1) all the unknown initial values in \( \text{rc} \) are instantiated by heap-names and \( \text{rc} \) evaluates to true (rcl is Empty-context in this case), or (2) \( \text{rc} \) is instantiated into a relevant context rcl that involves only the rcl interface initial values of a single public interface method and \( \text{rc} \) can evaluate to true by making conservative assumptions about these interface initial values. For example, consider call site 8 in public interface method method2 in Figure 1. \([(\text{empty-context},a_{5,\text{init}},\text{next},a_{5,\text{init}},\text{next},a_{5,\text{init}}),a_{2,\text{init}},\text{next},a_{4,\text{init}}]\) ∈ \( \text{8.def-undef-bindings} \), where \( a_{5,\text{init}} \) is an interface initial value and \( a_{5,\text{init}},\text{next},a_{5,\text{init}} \) is an interface initial def. As a result, initialize-worklist-II-dua propagates this binding as it is to the entry node of method2.

process-worklist-II-dua (Figure 10) iteratively propagates the values of unknown initial def's to the entry nodes of methods. instantiate-def-under-constraint(\( m, \text{rc}, \text{var}, s \)) instantiates the unknown initial values in the rc with their concrete values computed at the entry node of \( m \) during Phase II-p$a$ta; if \( \text{var} \) is an unknown initial value occurring in \( \text{rc} \), instantiate-def-under-constraint uses only s as the concrete value of \( \text{var} \) during each instantiation (hence the suffix under-constraint). This is the reason why \( \text{var} \) is stored in a def-undef-binding. instantiate-def-under-constraint returns a set of instantiations of \( \text{rc} \) that can evaluate to true. Each instantiation in this set is either (1) Empty-context, meaning the \( \text{rc} \) evaluates to true for the instantiation, or (2) it is a relevant context that involves only the interface initial values of a single public interface method and it can evaluate to true by making conservative, worst-case assumptions about these interface initial values.
for each SCC in topological order
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 
[rc1, ⟨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, 
[rc1, ⟨var, def⟩, uidef1]) 
// uidef1.method is the target (of c) with 
// whose entry node node1 is associated 
add-to-soln-and-worklist-if-needed-II(uidef1.method, result)
}
}
}
} 

2. process-worklist-II-dua() { 
while worklist is not empty {
wl-node = delete from worklist 
/**/ 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, uidef2⟩, ⟨def⟩] ∈ c.def-uidef-bindings 
rc3 = rc1 ∧ rc2 
new-rcs = 
instantiate-rc-under-constraint(m, rc3, var, s) 
for each rc ∈ new-rcs 
reaching-def = [rc3, ⟨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 9: Transfer functions for catch entry and catch exit nodes
Figure 10: phase II
6 Phase III

In this phase unknown initial values and unknown initial defs in DUA-dfelm-c, DUA-dfelm-d and DUA-dfelm-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 \( (\text{obj}, \text{def}) \) is used to instantiate an unknown initial def \( \text{uidef1} \) in a DUA-dfelm-c \( [r,c,\langle \text{var1}, \text{uidef1}, \text{use-point}\rangle] \) or a DUA-dfelm-e \( [r,c,\langle \text{var1}, \text{uidef1}, \text{use-edge}\rangle] \), \( \text{uidef1} \) is replaced by \( \text{def} \) and \( \text{var1} \) is replaced by \( \text{obj} \). This pairing of \( \text{obj} \) and \( \text{def} \) increases precision by not mixing \( \text{def} \)'s and \( \text{obj} \)'s reaching the corresponding entry node from different paths. Those instantiations of DUA-dfelm 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. \( [r,c,\langle \text{var}, \text{def-point}, \text{use-point}\rangle] \), where \( \text{def-point} \) is a program point or an interface initial def. \( \text{use-point} \) is \( n \) and \( \text{var} \) 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. \( [r,c,\langle \text{exc-object}, \text{throw-point}, \text{catch-point}\rangle] \), where \( \text{throw-point} \) is a throw statement number, \( \text{catch-point} \) is \( n \) and \( \text{exc-object} \) is a heap-name or an interface initial value. This is used for def-uses between throws and catches.

3. \( [r,c,\langle \text{var}, \text{def-point}, \text{use-edge}\rangle] \), where either \( \text{var} \) is a condition node and \( \text{use-edge} \) is a decision edge out of \( \text{var} \) or \( \text{n} \) is a dynamically dispatched call site and \( \text{use-edge} \) is an interprocedural edge between \( n \) and one of its targets. \( \text{def-point} \) and \( \text{var} \) are same as in 1.

In all cases \( r \) is either \( \text{Empty-context} \) (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 \( \text{method2} \) in Figure 1. At program point 7, the DUA-dfelm-c \( \{[\emptyset, \langle (\text{A::init}\rangle, \langle \text{empty}\rangle, \langle \text{global1,1,7}\rangle) \} \) is instantiated to \( \{[\emptyset, \langle (\text{A::init}\rangle, \langle \text{empty}\rangle, \langle \text{global1,1,7}\rangle) \} \) because \( a_{1\text{init}} \) is an interface initial value and it a possible concrete value of \( a_{1\text{init}} \) 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.[Lam92].

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

Let \( \text{RelevantContextCover}_L \) be the set of relevant contexts of the def-use associations computed by Phase III (see Section 6) for a library \( L \). Let \( \text{RelevantContextCover}_M \) be the subset of \( \text{RelevantContextCover}_L \) such that each element of \( \text{RelevantContextCover}_M \) 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 \( \text{null} \) 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 \( \text{ConsistentRelContCover}_M \).
be the subset of RelevantContextCover_M such that each element of RelevantRelContCover_M is consistent with the specification of t. One strategy for the tester is to manually\(^5\) enhance t using each element of RelevantRelContCover_M and generate size-off(RelevantRelContCover_M) test cases (one test case per element of RelevantRelContCover_M) which can be executed separately. However, there is a more efficient alternative: first the tester can compute a reduced context cover ReducedCover of RelevantRelContCover_M, having as small a size as possible and then generate size-off(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) de-clutter of some def-use paths require that non-pointer variables have specific initial values and unknown initial defs have been instantiated.

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

\[
\text{ReducedCover} = \emptyset \\
\text{for all } s \in \text{RelevantRelContCover}_M \\
\quad \text{if } s \text{ contains a representative unknown initial value} \\
\quad \quad \text{ReducedCover} = \text{ReducedCover} \cup \{s\} \\
\quad \text{else} \\
\quad \quad \text{if } (\exists r \in \text{ReducedCover} \text{ such that } r \land s \text{ cannot be satisfied and } r \land s \text{ is consistent with } t) \\
\quad \quad \quad \text{new-el} \ast \land r \land s \\
\quad \quad \quad \text{if ReducedCover} = \text{ReducedCover} \land \{\text{new-el}\} \\
\quad \quad \quad \text{
}\]

For example, RelevantContextCover_{example} for L_{example} given in Figure 1 is \{\{empty\}, \{a_{init} eq a_{init}\}, \{a_{init} neq a_{init}\}, \{(type)(a_{init}) \in A::update\}, \{(type)(a_{init}) \in C::update\}) and RelevantRelContCover_{method1} is same as RelevantContextCover_{method2}. As a result, the reduced context cover ReducedCover computed by the algorithm given above is \{\{a_{init} eq a_{init}\}, \{(type)(a_{init}) \in A::update\}, \{(type)(a_{init}) \in C::update\}\}.

\[\text{8 Discussion}\]

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 ExecObjTypeCont’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 ExecObjTypeCont’s provide a measure of code complexity. Complicated alias contexts, type contexts or ExecObjTypeCont’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

\[\text{\textsuperscript{5}It is possible to automate this step to a great extent and we plan to investigate this in future.} \]
6:1 \[ ([a_{init} \text{ eq } a_{2init}, \text{empty,empty}), (a_{2init}.next,4,6)] \]
6:2 \[ ([a_{init} \text{ neq } a_{2init}, \text{empty,empty}), (a_{2init}.next,a_{2init}.nextinitdef,6)] \]
7:1 \[ ([\text{empty,(type}(a_{init}) \in A::\text{update}), \text{empty}), (\text{global}1,1,7)] \]
7:2 \[ ([\text{empty,(type}(a_{init}) \in C::\text{update}), \text{empty}), (\text{global}1,2,7)] \]
17:1 \[ ([\text{empty,empty,(type}(e_{1init}) \leq ET2), \text{empty}), (\text{global}3,13,17)] \]
25:1 \[ ([\text{empty,empty,(type}(e_{2init}) \leq ET4), \text{empty}), (\text{global}3,13,25)] \]
27:1 \[ ([\text{empty,empty,(type}(e_{2init}) \not\leq ET2) \land (\text{type}(e_{2init}) \not\leq ET4), \text{empty}), (\text{global}3,13,27)] \]

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

/\*/\*\* interface initial values \text{a}_{4init} and \text{a}_{5init} are respectively concrete values of \text{a}_{linit} and \text{a}_{2init} at call site 8, and interface initial def \text{a}_{init}.nextinitdef is the concrete value of \text{a}_{2init}.nextinitdef at call site 8 */\*\*/
6:1 \[ ([a_{4init} \text{ eq } a_{5init}, \text{empty,empty}), (a_{5init}.next,4,6)] \]
6:2 \[ ([a_{4init} \text{ neq } a_{5init}, \text{empty,empty}), (a_{5init}.next,a_{5init}.nextinitdef,6)] \]
7:1 \[ ([\text{empty,(type}(a_{4init}) \in A::\text{update}), \text{empty}), (\text{global}1,1,7)] \]
7:2 \[ ([\text{empty,(type}(a_{4init}) \in C::\text{update}), \text{empty}), (\text{global}1,2,7)] \]
/\*/\*\* interface initial value \text{e}_{2init} is the concrete value of \text{e}_{linit} at call site 23 */\*\*/
17:1 \[ ([\text{empty,empty,(type}(e_{2init}) \leq ET2), \text{empty}), (\text{global}3,13,17)] \]
25:1 \[ ([\text{empty,empty,(type}(e_{2init}) \leq ET4), \text{empty}), (\text{global}3,13,25)] \]
27:1 \[ ([\text{empty,empty,(type}(e_{2init}) \not\leq ET2) \land (\text{type}(e_{2init}) \not\leq ET4), \text{empty}), (\text{global}3,13,27)] \]

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


A Auxiliary functions of init-worklist-l-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 = add-to-soln-and-worklist-if-needed-I(n.successor, y)  
   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 = add-to-soln-and-worklist-if-needed-I(n.successor, y)  
   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}  
   add-to-soln-and-worklist-if-needed-I(n.successor, gen-DUA-dfelms-a)  
   }

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}  
   add-to-soln-and-worklist-if-needed-I(n.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,objectw))  
   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-dfelms-a [excp-def,rc1,(var,def,(rc2,val))] or  
   dfe is DUA-dfelms-b [excp-def,rc1,excp-obj,throw-point] ****/  
   if ((dfe is a DUA-dfelms-b) or  
   (var is not a local variable and  
   def is a program point))  
   z = back-bind(dfe, c, m.exit-node)  
   /*** back-bind returns  
   [excp-def,rc1,excp-obj,throw-point]’s  
   to which dfe maps at c ***/  
   z = z  
   for each DUA-dfelms-a dfe ∈ x  
   z = z ∪  
   add-to-soln-and-worklist-if-needed-I(n.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

\[\text{rhs-rc-loc-pairs} = \{Empty-context, \text{don't-care}\}.\]

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,init.next\}\} and rhs-rc-loc-pairs is \{\{Empty-context,a_{1\text{init}}\}\}. Similarly, at statement 6 in Figure 1, lhs-rc-loc-pairs is \{\{Empty-context,global\}\} and rhs-rc-loc-pairs is \{\{\(\langle a_{1\text{init}} \text{ eq } a_2\rangle\), empty, empty, a_{3\text{init}}\}\}. This is because statement 4 modifies a_{2\text{init}.next} if and only if a_{1\text{init}} and a_{2\text{init}} are the same object at the entry of method1, and it does not modify a_{2\text{init}.next} if and only if a_{1\text{init}} and a_{2\text{init}} are not the same object at the entry of method1. As a result, Phase I-pta computes the PTA-dfels-a \{empty,\(\langle a_{1\text{init}} \text{ eq } a_2\rangle\), empty, empty\}, \(\langle a_{2\text{init}.next, a_{3\text{init}}}\rangle\} and \{empty,\(\langle a_{1\text{init}} \text{ neq } a_2\rangle\), empty, empty\}, \(\langle a_{2\text{init}.next, a_{3\text{init}.next}\rangle}\}\} on the top of statement 6.

\[\text{back-bind} (\text{dfe}, c, n)\] 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-dfels resulting from the instantiations.

### B generate-dfels-due-to-pot-aliases

\[\text{generate-dfels-due-to-pot-aliases}(n, \text{rdfe})\] generates DUA-dfels-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-dfels-due-to-pot-aliases generates the DUA-dfels-a \{empty, \(\langle a_{1\text{init}} \text{ eq } a_2\rangle\), empty, empty\}, \(\langle a_{2\text{init}.next, a_{3\text{init}}}\rangle\}

\[\text{back-bind} (\text{dfe}, c, n)\] 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-dfels resulting from the instantiations.

### C Auxiliary functions for process-call

\[\text{compute-p-uses-due-to-dynamic-dispatch}(n, \text{rdfe})\] computes the set of new-rc and uiv2.f for each possible target t with val as receiver. cond is the boolean condition of the dispatch. if (cond) new-rc = \text{rc1} ∧ \text{rc2} ∧ \langle typeof(val) \text{ compatible with } A \rangle

\[\text{compute-new-bindings-of-defs-and-u-i-defs}(n, \text{rdfe})\] computes the new-def-uidef-bindings \(\text{new-def-uidef-bindings} = \\{\langle \text{var,def},(n,t)\rangle\}\} if \(\text{var}\) is a local variable.

\[\text{compute-new-bindings-of-defs-and-u-i-defs}(n, \text{rdfe})\] computes the new-def-uidef-bindings \(\text{new-def-uidef-bindings} = \\{\langle \text{var,def},(n,t)\rangle\}\} if \(\text{var}\) is a global variable.

\[\text{compute-new-bindings-of-defs-and-u-i-defs}(n, \text{rdfe})\] computes the new-def-uidef-bindings \(\text{new-def-uidef-bindings} = \\{\langle \text{var,def},(n,t)\rangle\}\} if \(\text{var}\) is an object.

\[\text{compute-new-bindings-of-defs-and-u-i-defs}(n, \text{rdfe})\] computes the new-def-uidef-bindings \(\text{new-def-uidef-bindings} = \\{\langle \text{var,def},(n,t)\rangle\}\} if \(\text{var}\) is a method.

\[\text{generate-dfels-due-to-pot-aliases}(n, \text{rdfe})\] generates the DUA-dfels-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-dfels-due-to-pot-aliases generates the DUA-dfels-a \{empty, \(\langle a_{1\text{init}} \text{ eq } a_2\rangle\), empty, empty\}, \(\langle a_{2\text{init}.next, a_{3\text{init}}}\rangle\}

\[\text{back-bind} (\text{dfe}, c, n)\] 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-dfels resulting from the instantiations.
Figure 13: Return-site and method-exit-node transfer functions

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

\[ \text{back-bind}(\text{dfe}, c, n) \] uses the bindings in \( c\text{-actual-to-uv-bindings} \) and \( c\text{-def-aidef-bindings} \) to instantiate the unknown initial values and the unknown initial defs in \( \text{dfe} \) with their values at \( c \). \text{back-bind} returns the set of \( \text{DUA-dfelm} \) resulting from the instantiations.

Figure 14: Return statement transfer function