Impact Analysis Based on a Global Hierarchical Object Graph

Marwan Abi-Antoun Yibin Wang Ebrahim Khalaj Andrew Giang Vaclav Rajlich
Department of Computer Science, Wayne State University, Detroit, Michigan, USA

***Abstract*—During impact analysis on object-oriented code, statically extracting dependencies is often complicated by subclassing, programming to interfaces, aliasing, and collections, among others. When a tool recommends a large number of types or does not rank its recommendations, it may lead developers to explore more irrelevant code.**

We propose to mine and rank dependencies based on a global, hierarchical points-to graph that is extracted using abstract interpretation. A previous whole-program static analysis interprets a program enriched with annotations that express hierarchy, and over-approximates all the objects that may be created at runtime and how they may communicate. In this paper, an analysis mines the hierarchy and the edges in the graph to extract and rank dependencies such as the most important classes related to a class, or the most important classes behind an interface.

An evaluation using two case studies on two systems totaling 10,000 lines of code and five completed code modification tasks shows that following dependencies based on abstract interpretation achieves higher effectiveness compared to following dependencies extracted from the abstract syntax tree. As a result, developers explore less irrelevant code.

I. Introduction

Impact analysis is an important activity during software evolution [1]. During impact analysis, developers estimate what parts of the code may need to change if one part of the code were to change. To estimate those changes, they use dependencies between different parts of the code.

To compute those dependencies, several approaches have been proposed, ranging from static program analysis, dynamic program analysis, as well as analyses that take into account historical change information from a repository, among others.

Simple static analysis cannot expose some subtle dependencies that may exist between classes, such as ones due to the sharing of state, and these dependencies can be important during impact analysis. In particular, object-oriented code features such as inheritance, interfaces, collections, and possible aliasing, among others, tend to lower the quality of the results obtained from simple static analysis. If a static analysis conservatively estimates that everything is potentially impacted by one change, such a tool is not that useful.

As a result, statically extracted dependencies are often deemed too imprecise, so some tools use dynamic analysis [2]. The main limitation of dynamic analysis is that the results reflect only specific inputs or executions, and thus may not reflect dependencies that arise only for other inputs or other executions. As a result, the developers may miss some important dependencies.

Previous impact analysis tools based on static analysis used callgraphs, program slicing, static execute-after relationships, among others [3]. Instead of considering classes and computing dependencies based on visiting the program's Abstract Syntax Tree (AST), this work explores computing dependencies based on approximating what classes are created at runtime into *abstract objects*, then lifting that information back to classes. In particular, by approximating what abstract objects are created for a collection class, and to which abstract objects the collection object may point, one may be able to obtain more precise dependencies than by not distinguishing between different creations of collection classes, ignoring them, or assuming that the collection object may point to any abstract object. In program analysis terms, the underlying analysis is a whole-program static analysis that uses abstract interpretation and approximates the runtime structure as a global points-to graph that makes explicit what state is potentially shared.

More specifically, the underlying analysis extracts an Ownership Object Graph (OOG) [4]. The OOG is extracted using a whole-program analysis that analyzes the entire code and library code in use (or its summary). The OOG has abstract objects. Also, the OOG is sound in that it reflects all possible objects and the communication between them that may occur at runtime, in any possible execution, for any program input. Moreover, the OOG handles aliasing in that one object at runtime may not be represented by two distinct abstract objects in the graph, thus giving precision about shared state.

In an OOG, all abstract objects are organized into a global object hierarchy (single rooted tree), without filtering anything out. To express design intent, the OOG uses abstraction by hierarchy as follows: architecturally relevant objects from the application domain are at the higher levels of the object hierarchy, and implementation details such as data structures are at the lower levels. Since object hierarchy is missing in plain code, extracting an OOG requires adding to the code some annotations to express some of this design intent.

In this paper, we propose an analysis that mines the information in the OOG to extract and rank dependencies. Compared to the earlier work, this work de-emphasizes the OOG visualization; instead of showing abstract objects to developers, it lifts the information back to classes. The analysis uses the hierarchy in the OOG to rank dependencies and assign a lower rank to dependencies due to lower objects in the hierarchy. We then implement the analysis in a tool called ArchSummary, and evaluate it by comparing its dependencies on real code and for real code modification tasks to those of

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

SANER 2015, Montréal, Canada

JRipples [5], an open-source impact analysis tool. JRipples extracts static dependencies from the code without requiring annotations, and builds a Class and Member Dependency Graph (CMDG) based on visiting the program's AST.

The OOG also has abstract edges that track points-to relations between abstract objects, as well as dataflow communication based on the usage or flow of objects in the system due to field reads, field writes, or method invocations. To find the dependencies, the analysis traverses these edges then ranks the related classes based on the incoming or outgoing edges to or from an abstract object.

Contributions. The contributions of this paper are as follows:

- A definition of ranked dependencies for impact analysis based on abstract interpretation in terms of the most important classes related to a class, or the most important classes behind an interface;
- A detailed evaluation on two case studies on two systems and five completed code modification tasks. The evaluation consists of a thorough, step-by-step comparison of ArchSummary's recommendations to those of JRipples.

Outline. The rest of this paper is structured as follows. Section II gives some intuition behind the dependencies based on abstract interpretation and compares them to dependencies extracted from the AST. Section III defines the proposed dependencies. Section IV presents our evaluation method. Sections V-A and V-B present two case studies. Section VI discusses limitations and threats to validity. Next, we discuss related work (Section VII) and conclude.

II. Background

In this section, we first introduce a running example and show the CMDG that JRipples builds. We then compare the CMDG to the OOG and its internal representation, the OGraph. We then discuss the annotations and what they express, and how the abstract interpretation extracts an OGraph.

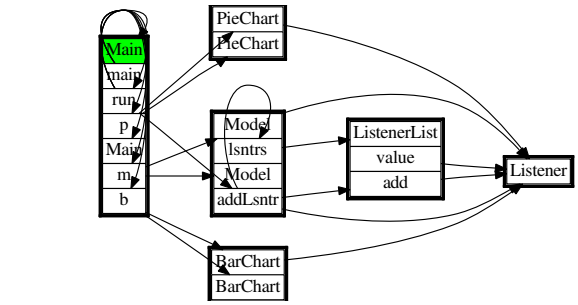
A. Motivating Example

As our motivating example, we use a small system, Listeners, which follows a Document-View architecture. In Listeners, a Model contains some data that is rendered graphically using a BarChart and a PieChart. The example illustrates a key part of the Observer design pattern, the adding of observers (listeners) to a list. We show the code with some annotations inside angle brackets <> (Fig. 2). We discuss the annotations later (Section II-D), so the reader may ignore them at first.

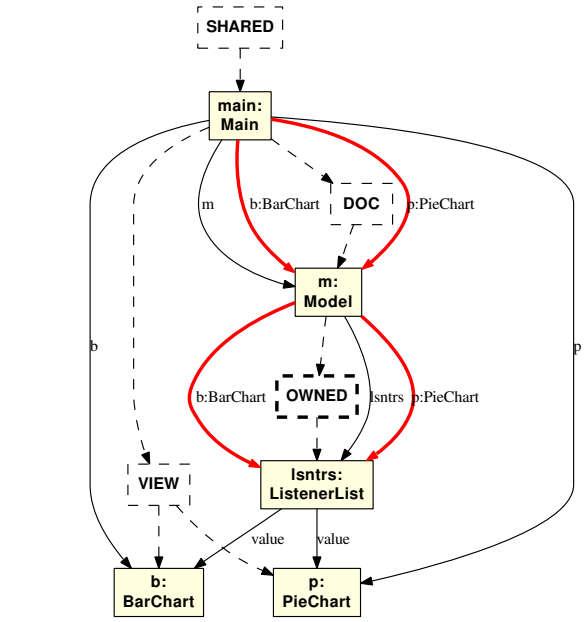
B. JRipples and Its CMDG

In a CMDG, the nodes represent the classes, methods and fields. The edges between the nodes in a CMDG are dependencies, of which there are two types: first, a field or a method of a class depends on its declaring class; and second, a node depends on another node if the code of the first node refers to the code of the second node.

We show a CMDG in Fig.1(a). The CMDG closely mirrors the code structure such as a class diagram; it shows classes and associations between them. From the CMDG of the Listeners example, the classes `Model`, `PieChart` and `BarChart` have a dependency on the interface `Listener`. But the CMDG does not clarify if at runtime, the corresponding objects of type `Model`, `PieChart` and `BarChart` share one `Listener` object. Similarly, if the code were to use the collection class `ListenerList` in many more places, many more classes will have a dependency on `ListenerList`, even if the corresponding objects do not share the same collection object at runtime. By contrast, the OGraph, which we discuss next, makes this sharing more explicit.



(a) CMDG.



(b) OGraph.

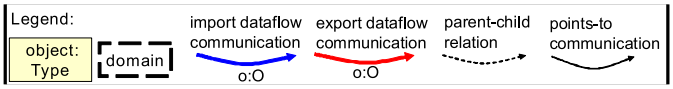


Fig. 1. Listeners CMDG vs. OGraph.

C. The OOG and Its Internal Representation, the OGraph

We show the OGraph in Fig. 1(b). Rather than showing classes like a class diagram, the OGraph shows abstract objects. One abstract object corresponds to zero or more runtime objects, abstracted by a (*type*, *domain*) pair, where *type* is the class of the object, and *domain* is a named conceptual group of objects. In particular, there may be multiple abstract objects of the same declared class if they are in different domains.

For example, the abstract object `m:Model` has its own `lstnrs:ListenerList` in domain `OWNED`, and this one is distinct from any other abstract objects of type `ListenerList` that is in another domain. Also, `lsntrs` points to `p:PieChart` and `b:BarChart`, but not back to `m:Model`—even though the class `Model` implements the `Listener` interface.

D. Extracting an OOG Using Annotations

To impose a hierarchy on the objects in the OOG, we add to the code type annotations that follow a type system called Ownership Domains [6]. The annotations express what object is the child of what other object, a fact that is not directly visible in plain object-oriented code. Instead of objects having have child objects directly, there is an extra level of indirection, a *domain*, which is a named, conceptual group of objects. Each object is in a single domain that does not change. Moreover, an object can have zero or more domains, and each domain can in turn have zero or more child objects, thus achieving an arbitrary object hierarchy based on design intent.

In this paper, we show the code with the enriched types as language extensions. The implementation, however, uses available language support for annotations, which leads to verbose constructs, but keeps the program fully compatible with existing tools. A typechecker for the Ownership Domains type system checks that the annotations are consistent with each other and with the code.

Domain names are chosen to convey design intent. For Listeners, we use `DOC` and `VIEW` to indicate a Document-View architecture. We show domain names in capital letters to distinguish them from other program identifiers. The domain names do not matter, however. What matters is that objects that are in the same domain may alias and may be merged by the static analysis, but objects that are in different domains may not alias and thus are kept distinct by the analysis.

Each class can declare one or more domains to hold the objects that make its parts using the `domain` keyword and a visibility modifier `private` or `public`. Next, we place an object in a domain as follows. In this type system, the owner of an object is expressed as part of its type. So each class takes a domain parameter called `owner`, shown between angle brackets <>, in the same manner as generic type parameters (generics are also supported but are not essential to this discussion so they are omitted). For example, the type of a field must now provide the owning domain for the object of that type. In the class `ListenerList`, which is a list of objects of type `Listener`, we model that the list has references to the elements it holds using the virtual field `value` of type `Listener<ELTS>`, meaning that the `Listener` objects are in the `ELTS` domain (Fig. 2, line 4).

Because a list class is intended to be reusable and able to hold objects in different domains depending on the client code, the class declares a domain parameter for the list elements. This domain parameter, `ELTS`, shown after the `owner` domain (line 3), and just like `owner`, can be bound differently depending on the client.

```
1  interface Listener<owner> {
2  }
3  class ListenerList<owner, ELTS> {
4    Listener<ELTS> value; // Virtual field
5    void add(Listener<ELTS> el) { ... }
6  }
7  class BarChart<owner, M> implements Listener<owner> {
8  }
9  class PieChart<owner, M> implements Listener<owner> {
10 }
11 class Model<owner, V> implements Listener<owner> {
12   private domain OWNED;
13   ListenerList<OWNED, V> lstnrs = new ...
14
15   void addLsntr(Listener<V> lstnr) {
16     lsntrs.add(lstnr);
17   }
18 }
19 class Main {
20   domain DOC, VIEW; // Top-level domains
21   Model<DOC, VIEW> m = new Model<...>();
22   BarChart<VIEW, DOC> b = new BarChart<...>();
23   PieChart<VIEW, DOC> p = new PieChart<...>();
24   void run() {
25     m.addLsntr(b);
26     m.addLsntr(p);
27   }
28 }
```

Fig. 2. Listeners: code with annotations.

More generally, a class C can declare an additional formal domain parameter α to access an additional domain, i.e., C becomes $C<\texttt{owner}, \alpha>$. Domain parameters allow objects to share state or access objects in other domains as follows. An object o of type C can access objects in a domain d of another object by *binding* that α to d. When instantiating a class that takes domain parameters, each domain parameter must be bound to another domain or domain parameter that is in scope. For instance, class `Model` needs to access objects in the `VIEW` domain so it takes a domain parameter `V`. In class `Main`, the field declaration `m` of type `Model` binds the `V` domain parameter to the `VIEW` domain (line 21).

A *private* domain holds internal state and provides *strict encapsulation*. For example, since the private domain `OWNED` (line 12 in Fig. 2) holds the `lstnrs` object, client code cannot access the `lstnrs` child object even if they have a reference to the `model` parent object—even if the `lstnrs` field does not have the standard `private` visibility modifier.

A *public* domain holds externally-visible state and expresses *logical containment* only, i.e., having access to an object means having access to all the objects inside its public domains. Public domains can thus express arbitrary design intent, and make it possible to group conceptually related objects. By placing an object a in the public domain of another object b, a becomes conceptually part of b, and the object graph has fewer objects that are at the same level as b. In the graph, a appears as a child of b but access to a is not restricted. Graphically, we show a private domain with a thick dashed border, and a public domain with a thin dashed border.

The OOG extraction analysis assumes that the program operates by creating a main object, an instance of the root class. We refer to the domains declared by the class of the root object as the *top-level domains*.

This section reviews the semantics of the OGraph, how the

$D \in \text{ODomain} ::= \langle \textbf{Id} = D_{id}, \textbf{Domain} = C{::}d \rangle$

$O \in \text{OObject} ::= \langle \textbf{Type} = C, \textbf{OwningDomain} = D1, \textbf{OtherDomain} = D2 \rangle$

$E \in \text{OEdge} ::= \langle \textbf{From} = O_{src}, \textbf{To} = O_{dst}, \textbf{Label} = L, \textbf{EdgeType} = ET \rangle$

Fig. 3. Key data type declarations for the OGraph.

abstract interpretation analyzes the code with annotations, and how it handles inheritance and interfaces, to better understand the differences with the JRipples analysis.

Data types. An OGraph (Fig. 3) is a graph with two types of nodes, OObjects referred to by the meta-variable O, and ODomains referred to by the meta-variable D. An OEdge, referred to by the meta-variable E, corresponds to a directed edge between a source OObject and destination OObject. It also has a label and a type (points-to, dataflow, etc.).

An OObject O is represented using the tuple $\langle C, D1, D2 \rangle$ to mean an abstract object of type C in ODomain $D1$, the tuple's second component (OwningDomain). Thus, in the object hierarchy, the OObject $\langle C, D1, D2 \rangle$ is shown as the child of that ODomain $D1$, which in turn is the child of some other OObject. The OGraph has a single root object. The OtherDomain, $D2$, identifies the OObject O further, and conceptually is a domain that the object has access to, i.e., O references objects from that domain $D2$.

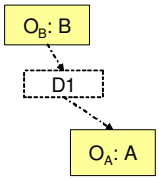
By having abstract objects of the form $\langle C, D1, D2 \rangle$, the OOG can distinguish between different abstract objects of the same class C that are in different OwningDomains, even if created at the same object creation expression. Also, two abstract objects can have the same OwningDomain but different OtherDomains. This situation occurs, for example, when two collection objects are in the same owning domain but store their elements in different domains. One domain declaration d in a class C can correspond to multiple ODomains D_i in the OGraph, where each D_i gets a fresh identifier D_{id}.

The analysis interprets an object creation expression (Fig. 4, line 11) as follows. The first formal domain parameter p_{owner} is the owning domain, and α_C is an additional formal domain parameter. At the creation site, each formal domain parameter must be bound to some other domain.

The analysis tracks the bindings of each domain in the code and maps it to an ODomain D_i in the OGraph. In particular, in some analysis context O_C, the analysis binds p_{owner} to some ODomain D_1, where D_1 is the child domain of some OObject O_B that has type B and its own actual domains.

Handling inheritance and interfaces. In the code (Fig. 4), the class C extends from the abstract class AC and implements the interface IC. So the abstract interpretation must handle inheritance and interfaces. Domain parameters are bound between a class and its super-class or super-interfaces. When analyzing a class, the analysis recursively traverses each super-class and super-interface, creating additional objects, domains or edges in the process. So the analysis takes into account inherited fields and methods that may contribute nodes or edges to the OGraph, but does not represent a super-class or super-interface as a separate node in the OGraph.

How JRipples handles inheritance and interfaces. In contrast, JRipples will show both a class C, its superclass AC and




```
// 5. analyze in same context as the implementing class                (1)
interface IC<owner> {...}                                              (2)
// 4. analyze in same context as the concrete subclass                 (3)
abstract class AC<owner> {...}                                         (4)
[ this ↦ O_C ] // 1. Track receiver, analysis context                  (5)
class C < owner, α_C > extends AC<owner>                               (6)
                              implements IC<owner> {                   (7)
  domain d_local;                                                      (8)
  // 2. Create ODomain  D_local = ⟨o.d, C :: d_local⟩                  (9)
  void m() {                                                           (10)
    new A<p_owner, α_C>();                                             (11)
    // 3. Create OObject  O_A = ⟨A, D_1, D_2⟩                         (12)
  }                                                                    (13)
}                                                                      (14)
```

Fig. 4. Abstract interpretation of `new` $A()$ in class C.

implemented superinterfaces IC_i, and show one dependency between C and AC, and one between C and each IC_i. For the Listeners example, JRipples shows the classes `Model`, `BarChart`, `PieChart`, and the interface `Listener` as separate nodes in the CMDG (Fig. 1(a)).

III. Approach

In this section, we discuss an analysis that lifts dependency information from the OGraph back to classes, and supports an impact analysis tool that developers use in an Integrated Development Environment while browsing classes.

We define the following dependencies:

- Most Important Classes (MICs);
- Most Important Methods of a Class (MIMs);
- Most Important Classes Related to a Class (MIRCs); and
- Most Important Classes Behind an Interface (MCBIs).

A. Most Important Classes: MICs

In an OGraph, architecturally significant objects from the application domain appear at the higher levels of the hierarchy, and implementation details such as data structures that are less architecturally significant appear at the lower levels. To compute MICs of a project, we use this hierarchy and include all the types of the objects that are defined in the top-level domains and the types of the objects that are in the public domains of those objects. A domain that is declared on the root class is a top-level domain.

$$MICs = \bigcup \{C_i\} \ \forall\ o_i{:}C_i \in D_i \text{ and } D_i \in domains(o_r)$$
$$\cup \bigcup \{C_j\} \ \forall\ o_j{:}C_j \in D_j \text{ and } D_j \in domains(o_i) \text{ and } public(D_j)$$

To find the MICs, the analysis finds all objects o_i of type C_i that are inside the domain D_i and D_i is a domain of the root object o_r, and adds C_i to MICs. Then for each o_i, it finds all the objects o_j of type C_j where o_j is in a public domain D_j of o_i. The MICs exclude the type of the root object.

In Fig. 1(b), the object `main` of type `Main` is the root object that also has two domains, `DOC` and `VIEW`. The object `m` of type `Model` is in `DOC` and the objects `p` of type `PieChart` and `b` of type `BarChart` are in `VIEW`. `DOC` and `VIEW` are top-level domains in this OGraph, so the types of the objects, `Model`, `PieChart`, and `BarChart` are added to the MICs.

Next, the analysis ranks the classes that are in the MICs according to the number of incoming and outgoing edges of all the abstract objects of a type in the OGraph. Because there may be multiple abstract objects of the same class, the analysis considers all of them.

We define a helper function $count$ that counts the number of incoming edges to an object of type C and outgoing edges from an object of type C. However, when looking for an object of type C, we also consider cases where we find objects of type C' where C' is a subtype of C, denoted using $<$:

$count(C) =$ number of e_i where
$\forall\ e_i\ \exists\ o_j{:}C_j$ where $C'{<}{:}C$ *// support subtyping*
$e_i = \langle o{:}C', o_j{:}C_j, L, ET\rangle$ or $e_i = \langle o_j{:}C_j, o{:}C', L, ET\rangle$
So $rank(C_i) < rank(C_j) \iff \sum count(C_i) < \sum count(C_j)$

According to the formal definition of the ranking strategy, the analysis counts all the edges e_i for each object o_i of type C_i that is included in the MICs. An edge e_i is defined as a tuple $\langle o : C, o'{:}C', L, ET\rangle$ where o is the origin object of type C, o' is the destination object of type C', L is the edge label and ET is the edge type. By computing the summation of all the edges for all the objects of the same type, a class with higher value will rank higher.

B. Most Important Methods of a Class C: MIMs(C)

Given a class C, the analysis computes MIMs(C) by adding method m, which is defined in the body of class C and creates edges in the OGraph, to MIMs(C). We define the $methods(C)$ function to return the list of all the methods declared in a given class C. We also define the function $traceability(e)$, returns the method that results in an edge e being created due to an expression such as a field read, field write or method invocation in the method declaration.

$MIMs(C) = \bigcup\{C'{::}m_i\}$
$\forall\ o{:}C'$ where $C'{<}{:}C$
$\forall\ e_i = \langle o{:}C', o_i{:}C_i, L, ET\rangle$ and
$C'{::}m_i \in methods(C')$ and $C'{::}m_i \in traceability(e_i)$

The analysis computes MIMs(C) by finding all the edges e_i that have o of type C', which is a subtype of C, as the origin object. It adds $C'{::}m_i$ to MIMs(C) if m_i is in the list of methods of C' and e_i is created because of an expression in m_i. In the base case, C and C' are the same and the analysis looks only for methods that are declared in class C.

In Fig. 1(b), two edges from the object `m` of type `Model` to the object `lsntrs` of type `ListenerList` labeled with `p:PieChart` and `b:BarChart` are created because of invoking the method `add` of the object `lsntrs` inside of the `addLsntr` method by passing the arguments `p` and `b`, respectively. The object `m` is the origin object of those edges and the `addLsntr` method is in $methods$(`Model`). So the `addLsntr` method is added to the MIMs(`Model`).

C. Most Important Classes Related to a Class: MIRCs(C)

To compute MIRCs(C) of a given class C, the analysis finds the most important related classes of class C according to the communication between objects of type C with other abstract objects. The communication is modeled as abstract edges in the OGraph. So the analysis finds all the outgoing edges from an object of type C and includes the type of the destination object into MIRCs(C). Also it finds all the incoming edges to an object of type C and adds the type of the origin object to the list of MIRCs(C).

$MIRCs(C) = \forall\ o{:}C'$ where $C'{<}{:}C$
$\bigcup\{C_i, C_j\}\ \forall\ e_i = \langle o{:}C', o_i{:}C_i, L, ET\rangle$
and $\forall\ e_j = \langle o_j{:}C_j, o{:}C', L, ET\rangle$

Given class C, for each object o of type C' that is a subtype of C, and for each outgoing edge of the object o, the analysis adds to the list of MIRCs(C) the type of the destination object. Then for each incoming edge, it adds the type of the origin object. In the base case, C and C' are the same.

In Fig. 1(b), to compute MIRCs(`Model`), the analysis finds all the objects of type `Model` or its subtypes in the OGraph. There is only one instance of class `Model` in this example, which is `m`. Then for each edge that has `m` as its origin, the analysis adds the type of the destination object of the edge to the list of MIRCs(`Model`). There are three edges from `m` to `lsntrs` of type `ListenerList`, so `ListenerList` is added to MIRCs(`Model`). Also, for each edge that has `m` as its destination, the type of the origin object is added to MIRCs(`Model`). There are three edges from `main` of type `Main` to `m`, so `Main` is added to MIRCs(`Model`).

D. Most Important Classes Behind an Interface: MCBIs(Tf)

If a program follows the recommended practice of programming to interfaces rather than implementation classes, it may have many references (fields, method parameters, and local variables) that are of interface types. During code modifications, a developer has to often find a class that implements that interface as the runtime type of the object that the reference may point to. Typically, more than one class implement an interface, but the true runtime type is a smaller set of classes. There is limited tool support to find the precise set of classes behind an interface, however. For example, the Eclipse Type Hierarchy returns all the classes that implement an interface.

For example, inside the class `Model` and method `addLstnr`, from looking at the method parameter `lstnr` of interface type `Listener`, it is not easy to tell what object is being registered as a listener. `BarChart`, `PieChart`, and `Model` all implement the `Listener` interface. Using MCBIs, the analysis uses the `V` domain annotation on `lstnr` and its type `Listener`, and finds only abstract objects of a type that is sub-type compatible with `Listener` and that are in domains that are bound to `V`, namely the objects of type `BarChart` and `PieChart` that in

the `VIEW` domain. As a result, MCBIs(`Listener<V> lstnr`) = {`BarChart`, `PieChart`}.

The formal definition of MCBIs requires a function $lookup$. Given a field Tf in class C where $T = C'<p', q'>$:

$$lookup\ (O_C, C'<p', q'>) = \bigcup\{C_k\} \text{ where}$$
$$D_1 = mapFtoA(O_C, p') \text{ // map domain in the code to ODomain}$$
$$D_2 = mapFtoA(O_C, q') \text{ // map domain in the code to ODomain}$$
$$\forall O_k = \langle C_k, D_1, D_2\rangle \in \mathsf{OObjects} \text{ where } C_k <: C'$$

For a field f of type $T = C'<p', q'>$, $lookup$ finds all the classes C_k for all the objects O_k of type C_k where C_k is a subtype of C' and O_k has the domains D_1, D_2 that p', q' map to, respectively, in some analysis context O_C. For a field declaration Tf, MCBIs unions the results of $lookup$ across all analysis contexts O_C:

$$MCBIs(T\ f) = \bigcup lookup(O_C, C'<p', q'>),\ \forall O_C$$

IV. Evaluation Method

To evaluate the approach, we conducted case studies to compare the effectiveness of dependencies extracted from an OGraph (ArchSummary) to those extracted from the AST (JRipples). During each case study, we completed code modification tasks and compared the types recommended by each tool during each task, or each step of completing the task.

A. Method

Evaluators. Two graduate students, the paper's second and third co-authors, completed coding tasks on the subject systems while using ArchSummary and JRipples. Neither evaluator was involved in the process of adding annotations or extracting OOGs from the subject systems, so they had no prior knowledge of the system design. Also, the evaluators did not design the approaches nor implement the tools being evaluated (ArchSummary or JRipples); they learned how to use them after the fact. Moreover, several characteristics of the approach and the evaluation lower the threat to validity of using co-authors as evaluators to compare the tools's recommendations. The annotations are checked by a type-checker and are under version control (they were not changed). Any mismatch between the annotations and the code leads to a visible warning in the Eclipse Problems window. The OOG extraction analysis does not filter objects out so the evaluators cannot change the OOG except by changing the annotations. The outputs of the tools are read-only, i.e., MICs or MIRCs cannot be edited or reordered. Finally, the tool's recommendations were logged as we discuss below.

Tools. The evaluators used Eclipse 4.2. The ArchSummary evaluator used the ArchSummary plugin, which shows the MICs, MIRCs, MIMs and MBCIs as Eclipse views. The JRipples evaluator used the JRipples plugin (version 3.2.2).

Instrumentation. We instrumented both JRipples and ArchSummary with a logging capability that records the types being recommended, being visited, etc., at each *step*. A *step* is defined as each time the tool recommends a (possibly empty) set of types.

The logging enables us to accurately and fully capture the output of the tools, as opposed to recording the screen then transcribing the recording manually. We fine-tuned our logging as follows. ArchSummary's MICs and MIRCs may contain library classes, including collections. JRipples, however, filters out from its output library classes since developers typically do not modify them. Moreover, collections can be a source of imprecision for static analysis so JRipples filters them out. To make the data comparable, the logging in ArchSummary filters out library classes, but their impact is still considered by the tool. Also, since MIMs in ArchSummary are at a finer level of granularity than types, so the logging does not record them. Finally, we make the logs available in an appendix [8].

B. ArchSummary Procedure

Developers using ArchSummary for impact analysis usually start from the top-ranked classes in MICs. For any type that developers explore for the first time in Eclipse, ArchSummary automatically marks it as Visited. After examining a recommended class, developers replace the mark with Impacted if the class needs to change, or Unchanged otherwise. As soon as the developers pick a class C from MICs, they are free to go to any of the following and jump between them freely:

1) **MICs:** developers pick a class C that does not have a mark but with a higher rank, explore it in Eclipse editor;
2) **MIRCs(C):** when developers open a class C, MIRCs get updated based on this class. Developers can pick a new class in MIRCs that does not have a mark; they typically consider the rank and start with higher ranked classes;
3) **MIMs(C):** developers may look at the MIMs as a summary of a class C, focusing on the methods that contribute dependencies;
4) **MCBIs(T f):** when encountering a field f declared with a type T (interface or abstract class) in class C, developers look at the MCBIs for that field. Based on the information, they pick a class that has not been visited while considering its rank.

By the construction of the OOG, interfaces or abstract classes do not appear in ArchSummary's recommendations. When developers are inspecting a class, they see its superclass or implemented superinterfaces. If developers wish to explore all possible subtypes of a class or interface, they can use the Eclipse Type Hierarchy feature.

C. JRipples Procedure

The procedure to use JRipples is a bit more constrained and is discussed elsewhere [7]. Briefly, impact analysis in JRipples can be divided into two phases: First, the developers select and visit the starting point of the project which is always the main class. JRipples marks every neighbor of the visited type in the CMDG with a Next mark as a recommendation. After the developers examine a new type marked as Next, they decide which of the following marks to apply to this type: Located, Propagating, or Unchanged. The mark Located means the developers intend to change the type. The mark Unchanged means the developers do not intend to change the type. Both of

these marks do not lead to new recommendations in JRipples. When the developers set the mark Propagating, they do not intend to change this type but its unvisited neighbors may need to change. Setting this mark leads JRipples to mark new types as Next for further inspection by the developers.

In JRipples, developers keep visiting recommendations until no further type seems related to the current task, based on its name or package. Then they decide to estimate the impact of types which are marked as Located and compose the initial *Impacted Set*. JRipples replaces the mark Located with Impacted automatically for the initial Impacted Set and uses these types as starting points in the next phase. The second phase has a similar procedure like the former one and almost the same marks (Propagating, Unchanged, Next, Impacted). The differences are that the mark Impacted represents that the marked type is intended to be changed and it leads to unvisited neighbors of this type to be marked as Next by JRipples. All recommendations in this phase should be inspected by the developers. After the two phases above, the developers finish impact analysis with a set of types marked as Impacted.

Table I shows the correspondence between the marks as used in ArchSummary and JRipples. We account for the differences in computing the measures below.

D. Hypothesis and Measures

We propose the following hypothesis:

Following dependencies based on abstract interpretation leads to higher effectiveness in impact analysis compared to following dependencies based on the AST, where effectiveness is defined in terms of the impacted set and the visited types.

To test the hypothesis, we measure the following:

- **Distinct Recommended Types (DRT):** is the number of distinct types that each tool recommends, in aggregate, for the task;
- **Recommended Types per Step (RTS):** is the number of types that each tool recommends during a step;
- **Number of Visited Types (NVT):** is the number of the types that each evaluator actually visits to complete impact analysis in the given task. The types are included in the VisitedSet;
- **Effectiveness:** is the ratio of the number of true positives in the ImpactedSet and the number of visited types NVT. The ImpactedSet is the result of impact analysis and contains the types that the evaluator intends to change. The true positives in this set are the types that actually change after implementing the task. All the types that are actually changed form the ChangedSet of the task;
- **Tool-specific:** we also measure tool-specific outputs and their closest counterparts in the other tool: MCBIs vs. All_Types, MCBIs_Invoked vs. Interfaces_Visited.

We discuss each of the measures in more details below.

Distinct Recommended Types (DRT). To compute DRT for ArchSummary, we get the MICs of the project, the MIRCs of each visited class during the task, the MCBIs when the evaluator invokes MCBIs for a field declaration of an interface type, union them all to eliminate duplicates, and finally compute the size of the resulting set. In JRipples, DRT is the union of all the types that have a mark.

Recommended Types per Step (RTS). For ArchSummary, RTS is computed by measuring the set of recommended classes from MIRCs. For JRipples, RTS is the set of all neighbors of the type marked as Propagating or Impacted. Then we compute the average Avg and the maximum Max of RTS for each tool and for each task.

Number of Visited Types (NVT). Because we use marks for visited types in both tools, NVT is the number of all types that have any mark except Next in each task.

Effectiveness. The types marked as Impacted in ArchSummary and JRipples, respectively, form the ImpactedSet for each task. The types that are actually changed form the ChangedSet and are computed manually based on the modifications the evaluators make to the code after impact analysis

We then compute the true positives in the ImpactedSet, which we denote by TP Impacted as follows

$$\mathsf{TP\ Impacted} = \{\mathsf{ImpactedSet}\} \cap \{\mathsf{ChangedSet}\}$$

Effectiveness is the ratio of the size of TP Impacted and NVT:

$$\text{Effectiveness} = \frac{|\mathsf{TP\ Impacted}|}{\text{NVT}} \times 100\%$$

MCBIs vs. AllTypes. During a task, for each field declaration f of an interface type T on which the ArchSummary evaluator invokes MCBIs, we record MCBIs($T\,f$). Similarly, in JRipples, we compute All_Types(T'), which is the number of the subtypes of a visited interface T' obtained from the Eclipse Type Hierarchy. We then compute, per task, the average Avg and the maximum Max for MCBIs and All_Types, across the intersection of interfaces found in the logs of ArchSummary and JRipples. To clarify, JRipples itself can recommend only direct subtypes of T' combined with other types that have AST dependencies to T' if the evaluator marks T' as Impacted or Propagating, so we use the Eclipse Type Hierarchy to collect all subtypes of T'. Besides, the JRipples evaluator may visit some irrelevant interfaces when strictly following the JRipples procedure. Such differences make the set of interfaces in the logs of the two tools different, thus we compare MCBIs and All_Types for interfaces that are inspected by both evaluators for a more accurate comparison.

MCBIs_Invoked vs. Interfaces_Visited. For MCBIs and All_Types, we compare the recommendations of each tool for only the same interfaces. We are also interested in how many interfaces the ArchSummary evaluator inspects carefully, or the JRipples evaluator visits in each task. MCBIs_Invoked counts the distinct interface types of the field declarations on which the ArchSummary evaluator invokes MCBIs, whereas Interfaces_Visited counts all the interfaces in the JRipples VisitedSet.

V. Evaluation

In this section, we present two case studies on two subject systems, MiniDraw and DrawLets, and five completed code modification tasks, focusing on the quantitative and qualitative

TABLE I
MAPPING OF MARKS BETWEEN ARCHSUMMARY AND JRIPPLES.

Mark	ArchSummary	JRipples
Next	N/A	Next
Unchanged	Unchanged	Unchanged/Propagating
Impacted	Impacted	Impacted
Visited	Visited	Impacted/Unchanged/Propagating

comparison of the two types of dependencies. Due to space limits, the following are left in an online appendix [8]: (a) the details of adding annotations and extracting OOGs for the subject systems; (b) the description of the tasks (T1–T5); (c) the raw log files; and (d) the detailed reports of the navigation of the evaluators and the analysis of the raw data.

Table II and Table III show the results of the measures. Based on the final implementations of these tasks, T1 to T4 are small changes and T5 is medium-sized change (it impacts more than 10 types).

A. Case Study 1: MiniDraw (MD)

For the first study, we use MiniDraw (MD), an object-oriented framework that supports developing board games [9]. MD is about 1400 LOC and has 68 classes and interfaces. The tasks T1–T4 were conducted on MD. T1 adds validation of the piece movement on the board. T2 implements the capture of a board piece. T3 implements an undo feature for a piece movement on the board. T4 implements a status bar to be updated on each piece movement.

As shown in Table II, to complete T1, the NVT of ArchSummary is 5 but the JRipples NVT is 12. The comparisons are more striking for T2 (8 vs. 21), T3 (7 vs. 19) and T4 (8 vs. 18), which means the JRipples evaluator visits almost triple the number of types compared to the ArchSummary evaluator.

The Max and Avg for MCBIs and All_Types show the differences between ArchSummary and JRipples for the same interfaces and abstract classes. The numbers of MCBIs are always smaller than those of All_Types, so this is one area where the ArchSummary recommendations are more precise.

Furthermore, in ArchSummary, MCBIs allow developers to concentrate on the concrete classes implementing interfaces and abstract classes. During all the tasks, the ArchSummary evaluator invokes MCBIs for field declarations of distinct interfaces only 2 or 3 times (shown in MCBIs_Invoked) whereas the JRipples evaluator inspects up to 8 interfaces (shown in Interfaces_Visited) that are mostly irrelevant.

From the results, the ArchSummary DRT is always smaller, which means that, across the entire task, ArchSummary recommends fewer types compared to JRipples. Meanwhile, for RTS, the ArchSummary values are sometimes larger and sometimes smaller than the JRipples ones. This indicates that dependencies computed based on abstract interpretation lead to different results than those based on the AST.

To clarify, the ArchSummary Max RTS stays at 14 because for each task, the evaluator visits the same class, which recommends the greatest number of types. The ArchSummary DRT seems stable during all tasks (21 in T1 and T2, 22 in T3 and T4) because those impacted classes have high ranks and

TABLE II
COMPARATIVE RESULTS TO COMPLETE T1, T2, T3 AND T4.

	ArchSummary		JRipples	
T1	DRT	21	DRT	24
	RTS Avg / Max	7.4 / 14	RTS Avg / Max	2.3 / 8
	NVT	5	NVT	12
	Effectiveness	20%	Effectiveness	8%
	MCBIs Avg / Max	1 / 1	All_Types Avg / Max	2 / 2
	MCBIs_Invoked	3	Interfaces_Visited	4
T2	DRT	21	DRT	41
	RTS Avg / Max	6 / 14	RTS Avg / Max	8.4 / 17
	NVT	8	NVT	21
	Effectiveness	25%	Effectiveness	10%
	MCBIs Avg / Max	1 / 1	All_Types Avg / Max	2 / 2
	MCBIs_Invoked	2	Interfaces_Visited	8
T3	DRT	22	DRT	30
	RTS Avg / Max	7 / 14	RTS Avg / Max	6.4 / 10
	NVT	7	NVT	19
	Effectiveness	29%	Effectiveness	11%
	MCBIs Avg / Max	1 / 1	All_Types Avg / Max	2 / 2
	MCBIs_Invoked	2	Interfaces_Visited	8
T4	DRT	22	DRT	25
	RTS Avg / Max	6.5 / 14	RTS Avg / Max	8.7 / 15
	NVT	8	NVT	18
	Effectiveness	25%	Effectiveness	11%
	MCBIs Avg / Max	1 / 1	All_Types Avg / Max	2 / 2
	MCBIs_Invoked	3	Interfaces_Visited	7

can be found in a similar way. JRipples recommends nearly the same number of concrete types (approximately, it equals DRT minus Interfaces_Visited) or even fewer types (recall, JRipples recommends abstract types too), but has a much higher DRT especially for T2 and T3. This confirms that the evaluators struggle with interfaces and abstract classes.

For each task, both evaluators get ImpactedSets of the same size and 0 false positives. False negatives occur only in T3 which requires developers to add some new types along with modifying existing types. As a result, the Effectiveness of ArchSummary is higher than that of JRipples since ArchSummary has a small NVT across all tasks.

B. Case Study 2: DrawLets (DL)

For the second case study, we use DrawLets (DL) (version 2.0), which has 115 classes, 23 interfaces, 12 packages, and 8,800 lines of code, This system was previously studied by others [10], [11], [12]. The task T5 was conducted on DL. DL supports a drawing canvas that holds figures and lets users interact with them. For T5, the goal is to implement an "owner" for each figure, which impacts many classes.

In T5, many inspected types contain a large number of fields, methods, or both, which generate dependencies to new types. The JRipples NVT reaches 97 and most of the visited types actually have no relation to this task, which causes a low Effectiveness of 17%. In this case, the JRipples evaluator inspects many irrelevant types. For ArchSummary, the NVT decreases dramatically to 37. For T5, the ArchSummary Effectiveness is 35%, a good improvement over JRipples.

Although the Avg RTS is higher for ArchSummary than for JRipples, ArchSummary generates a far smaller DRT compared to JRipples because JRipples continuously adds new types while following the JRipples procedure.

Here again, the JRipples evaluator struggles with interfaces.

TABLE III
COMPARATIVE RESULTS TO COMPLETE T5.

	ArchSummary		JRipples	
T5	DRT	53	DRT	100
	RTS Avg / Max	23 / 46	RTS Avg / Max	17 / 58
	NVT	37	NVT	97
	Effectiveness	35%	Effectiveness	17%
	MCBIs Avg / Max	2.9 / 12	All_Types Avg / Max	6.5 / 19
	MCBIs_Invoked	8	Interfaces_Visited	20

After visiting a certain interface, all the types that have a dependency with this interface—not just its subtypes—are marked as Next and combined with other unvisited recommendations from earlier steps. For instance, after the JRipples evaluator visits one interface (called `Figure`), JRipples finds 58 neighbors, which is the Max for RTS. Besides, the JRipples evaluator recursively marks the children of the interface as Propagating or Impacted until he finds all the impacted subtypes. According to the Avg for All_Types, the JRipples evaluator inspects 6.5 types for each interface whereas the ArchSummary evaluator visits only 2.9 types for the same set of interfaces. The difference is caused by MCBIs filtering out subtypes that are in unreachable domains.

With ArchSummary, the evaluator pays attention only to the interfaces in which they are interested. From Table III, the ArchSummary evaluator inspects field declarations of 8 different interfaces, compared to the JRipples evaluator who visits 20 interfaces.

It is worth noting that after implementing the task T5 based on the ImpactedSet from each tool, we find that both predictions contain only 1 false negative which is a new added class specifically for this task and 0 false positives. However when we compare the ImpactedSets, the JRipples evaluator detects 3 more classes. These classes are not in any ArchSummary recommendations as they are not instantiated in the project, which means they are not used and thus do not exist in the OGraph. Besides, the missing classes are subtypes of some impacted classes found during the ArchSummary procedure, so its evaluator can use the Eclipse Type Hierarchy to discover the additional types.

VI. DISCUSSION

Why JRipples for comparison? JRipples is designed specifically to support impact analysis and capture the thought process of developers using the marks. Moreover, since the tool is open-source and extensible, other researchers not affiliated with our group are also using it to implement and compare various dependency algorithms [3].

Effects of navigation. The OOG extraction analysis is whole-program and flow-insensitive, so all navigations in ArchSummary produce the same results. The only way to change the results is to change the annotations, make sure they still typecheck, and re-extract the OOG. The marks the evaluator sets in ArchSummary do not affect the result, but record the evaluator's thought process and enable a meaningful comparison with JRipples. In contrast, in JRipples, when the evaluator sets the mark to Impacted, JRipples may show new dependencies since it marks all the unmarked, possibly impacted neighboring methods with the mark Next. To account for this, we measured each tool's cumulative recommendations across the entire task.

Annotation overhead. A previous study measured the effort of adding annotations manually to be 1 hour/KLOC [13], assuming that the evaluator learned what the annotations mean. So one can estimate the effort based on the system size. Recent work in inferring ownership types [14] is inferring many of these annotations automatically. In particular, private domains can be inferred automatically. Public domains, however, require some design intent. A previous empirical study on 100 KLOC of code with annotations, which includes the two systems used in our case studies, shows that private domains were used for 2%–16% of the annotations and that defaults can be used for 5%–45% of the annotations [15, Table 4]. So the remaining annotations have to be supplied semi-automatically. We also have tools to add initial annotations based on a user-specified map of types to domains [15].

A. Threats to Validity and Limitations

Specific strategies. In our case studies, we compared the AST-based dependencies to dependencies based on abstract interpretation based on one set of strategies to mine and rank the dependencies from the OOG. It is likely that other strategies may lead to different results, for example, if they take into account other types of edges such as creational edges. It is clear, however, that abstract interpretation leads to different results. Library classes, and in particular collections, contribute many interesting dependencies between application classes. Simply ignoring them as JRipples does is likely leaving out important potential dependencies that developers must be aware of. For Listeners (Fig. 1), the CMDG does not show a dependency between `addLstnr` in `Model` and `PieChart` and `BarChart`, like the OGraph does.

Limited evaluation. Another threat to validity is that our evaluation used relatively small systems. The systems, however, use many design patterns and are far from trivial. For example, several participants in a controlled experiment struggled with completing the same tasks on MiniDraw [16]. Our evaluators may not be representative of other developers. The study, however, measures the differences in the recommendations of two tools when evaluators systematically follow a procedure and set marks, rather than how quickly the evaluators complete the tasks on their own without using impact analysis tools.

Evaluators. One threat to validity is using co-authors for the evaluation. We militated this threat as discussed in Section IV. This paper contributes new types of dependencies and compares the recommendations generated by two tools rather than developer behavior. In future work, we will evaluate the tools in a controlled experiment.

Unsound extraction. Many programs rely on configuration files, reflection and dynamic code loading to achieve extensibility. But static analysis does not directly handle these features. For such a program, the OOG can be unsound. To mitigate some of this unsoundness, however, the effects of reflection can be summarized using additional annotations.

VII. Related Work

There are many approaches for impact analysis, and they vary based on the type of dependence graph they use and how they extract it using either static analysis, dynamic analysis, mining software repositories, textual information, or a combination of the above. We discuss each one in turn below.

Static analysis. There are several static impact analysis tools. Chianti [17] requires a suite of regression tests associated with a Java program and access to the original and edited versions of the code. ArchSummary requires adding annotations only. Dora [18] exploits both program structure and lexical information to prune irrelevant structure edges from consideration. It would interesting to see if a similar approach can be combined with the dependencies mined from the OOG. In particular, the OOG currently disregards the identifiers of the variables, which can be an important source of information. Since the OOG abstracts objects to pairs of types and domains, it relies only on the domain annotation, the type of a variable, and the subtyping relations declared at the level of the types.

Dynamic analysis. Some approaches extract a graph using dynamic analysis [19]; such a graph, while possibly much more precise than a statically extracted graph, is unsound and reflects only the possible relations triggered by the test suites and the test inputs. In particular, compared to the statically extracted graph which considers even infeasible paths, a dynamic analysis can be path-sensitive. A dynamic analysis can also readily handle reflection and dynamic code loading, which remain challenging for a static analysis.

Mining software repositories. Some approaches extract a dependency graph by mining software repositories [20]. This approach assumes the presence of rich historical change information. In some cases, this information may be missing. Many such approaches also require multiple sources of information. For instance, the approach by Gethers et al. [20] requires the code of one complete release of the system, the change history, and the execution and tracing environment

Textual information. Some approaches extract a dependency graph by using information retrieval on textual information that can be either the text of the source code or of artifacts external to the code such as change requests [21]. The results depend on the history of prior change requests; as a result, the technique may not work well without having a detailed change request history. Reconstructing a history after the fact can be problematic, but annotations can be added after the fact to use ArchSummary. Moreover, the annotations are checked using a tool to be consistent with the code and with each other.

Using OOGs during software evolution. Abi-Antoun and Ammar evaluated how developers use OOGs for coding tasks using a case study [12] then using a controlled experiment [16]. Both previous studies presented to developers OOGs as diagrams that complement class diagrams, and did not involve using MICs, MIRCs, etc.

VIII. Conclusion

In this paper, we studied static impact analysis based on a whole-program analysis combined with abstraction and strategies that control the ranking and the number of dependencies being shown. Our evaluation to date shows that such dependencies are more precise than dependencies extracted from traversing the program's abstract syntax tree. Unlike other approaches that require change history or information external to the code, this approach requires adding annotations to the code after the fact, so it can be combined with other approaches that use the code. In future work, we plan to explore additional strategies to mine and rank the dependencies, as well as broaden the comparative evaluation to consider more impact analysis tools.

Acknowledgements

This work is supported in part by the National Security Agency lablet contract #H98230-14-C-0140. The authors thank the anonymous reviewers, Radu Vanciu, Andrian Marcus and Laura Moreno for helpful comments and discussions.

References

[1] S. Bohner and R. Arnold, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[2] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE*, 2003, pp. 308–318.

[3] G. Tóth, P. Hegedűs, A. Beszédes, T. Gyimóthy, and J. Jász, "Comparison of different impact analysis methods and programmer's opinion: An empirical study," in *Intl. Conference on the Principles and Practice of Programming in Java*, 2010, pp. 109–118.

[4] M. Abi-Antoun and J. Aldrich, "Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations," in *OOPSLA*, 2009, pp. 321–340.

[5] M. Petrenko and V. Rajlich, "Variable Granularity for Improving Precision of Impact Analysis," in *ICPC*, 2009.

[6] J. Aldrich and C. Chambers, "Ownership Domains: Separating Aliasing Policy from Mechanism," in *ECOOP*, 2004, pp. 1–25.

[7] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "JRipples: a tool for program comprehension during incremental change," in *IWPC*, 2005.

[8] "Raw data," www.cs.wayne.edu/~mabianto/arch_summary/, 2014.

[9] H. B. Christensen, *Flexible, Reliable Software Using Patterns and Agile Development*. Chapman and Hall/CRC, 2010.

[10] V. Rajlich and P. Gosavi, "A Case Study of Unanticipated Incremental Change," in *ICSM*, 2002.

[11] M. Skoglund and P. Runeson, "A Case Study on Regression Test Suite Maintenance in System Evolution," in *ICSM*, 2004.

[12] M. Abi-Antoun and N. Ammar, "A Case Study in Evaluating the Usefulness of the Run-time Structure during Coding Tasks," in *Workshop on Human Aspects of Software Engineering (HAoSE)*, 2010, pp. 1–6.

[13] M. Abi-Antoun, N. Ammar, and Z. Hailat, "Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report," in *Quality of Software Architectures (QoSA)*, 2012, pp. 133–142.

[14] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst, "Inference and checking of object ownership," in *ECOOP*, 2012.

[15] R. Vanciu and M. Abi-Antoun, "Object Graphs with Ownership Domains: an Empirical Study," in *State-of-the-art Survey on Aliasing in Object-Oriented Programming*. Springer-Verlag, 2013, pp. 109–155.

[16] N. Ammar and M. Abi-Antoun, "Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks," in *WCRE*, 2012, pp. 367–376.

[17] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. C. Chesley, "Chianti: a Tool for Change Impact Analysis of Java Programs," in *OOPSLA*, 2004.

[18] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with Dora to expedite software maintenance," in *ASE*, 2007.

[19] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE*, 2003.

[20] M. Gethers, B. Dit, H. H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *ICSE*, 2012.

[21] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using Information Retrieval Based Coupling Measures for Impact Analysis," *Empirical Softw. Engg.*, vol. 14, no. 1, 2009.