Method-Level Program Dependence Abstraction and Its Application to Impact Analysis

Haipeng Cai

Washington State University, Pullman, WA; hcai@eecs.wsu.edu

Raul Santelices

Delphix, Atlanta, GA; rasantel@gmail.com

Abstract

The traditional software dependence (TSD) model based on the system dependence graph enables precise fine-grained program dependence analysis that supports a range of software analysis and testing tasks. However, this model often faces scalability challenges that hinder its applications as it can be unnecessarily expensive, especially for client analyses where coarser results suffice.

This paper revisits the static-execute-after (SEA), the most recent TSD abstraction approach, for its accuracy in approximating method-level forward dependencies relative to the TSD model. It also presents an alternative approach called the method dependence graph (MDG), compares its accuracy against the SEA, and explores applications of the dependence abstraction in the context of dependence-based impact analysis.

Unlike the SEA approach which roughly approximates dependencies via method-level control flows only, the MDG incorporates more fine-grained analyses of control and data dependencies to avoid being overly conservative. Meanwhile, the MDG avoids being overly expensive by ignoring context sensitivity in transitive interprocedural dependence computation and flow sensitivity in computing data dependencies induced by heap objects.

Our empirical studies revealed that (1) the MDG can approximate the TSD model safely, for method-level forward dependence at least, at much lower cost yet with low loss of precision, (2) for the same purpose, while both are safe and more efficient than the TSD model, the MDG can achieve higher precision than the SEA with better efficiency, both significantly, and (3) as example applications, the MDG can greatly enhance the cost-effectiveness of both static and dynamic impact analysis techniques that are based on program dependence analysis.

More generally, as a program dependence representation, the MDG provides a viable solution to many challenges that can be reduced to balancing cost and effectiveness faced by dependence-based tasks other than impact analysis.

Keywords: Dependence analysis, dependence abstraction, method dependence graph (MDG), impact analysis, accuracy, cost-effectiveness

---

1. Introduction

Program dependence analysis has long been underlying a wide range of software analysis and testing techniques (e.g., [1, 2, 3, 4]). While traditional approaches to dependence analysis offer fine-grained results (at statement or even instruction level) [5, 6], they can face severe scalability and/or usability challenges, especially with modern software of growing sizes and/or increasing complexity [7, 8], even more so when high precision is demanded with safety guarantee [9, 10].

On the other hand, for many software-engineering tasks where results of coarser granularity suffice, computing the finest-grained dependencies tends to be superfluous and ends up with low cost-effectiveness in particular application contexts—in this work, a (dependence) analysis is considered cost-effective (measured by the ratio of effectiveness to cost) if it produces effective (measured by accuracy, or precision alone if with constantly perfect recall) results relative to the total overhead it incurs (including analysis cost and human cost inspecting the analysis results) [11]. One example is impact analysis [12], which analyzes the effects of specific program components, or changes to them, on the rest of the program to support software evolution and many other client analyses, including regression testing [13, 14] and fault localization [15]. For such tasks as impact analysis, results are commonly given at method level [16, 17, 18], where fine (e.g., statement-level) results can be too large to fully utilize [8]. In other contexts such as program understanding, method-level results are also more practical to explore than those of the finest granularity.

Driven by varying needs, different approaches have been explored to abstract program dependencies to coarser levels, including the *program summary graph* [19] used to speed up interprocedural data-flow analysis, the *object-oriented class-member dependence graph* [20], the *lattice of class and method dependence* [21], the *influence graph* [22], that are all used for impact analysis. and the *module dependence graph* [23] used for understanding and improving software structure. While these abstractions have been shown useful for their particular client analyses, they either capture only partial dependencies

among methods [22, 20] or dependencies at levels of classes [21] even files [23], which can be overly coarse for many dependence-based tasks. More critically, most such approaches were not designed or fully evaluated as a general program dependence abstraction with respect to their accuracy (both precision and recall) against that of the original full model they approximate as ground truth.

Initially intended to replace traditional software dependencies (TSD) that are based on the system dependence graph (SDG) [6, 7], a method-level dependence abstraction, called the *static-execute-after/before* (SEA/SEB) [7], has been proposed recently. It abstracts dependencies among methods based on the interprocedural control flow graph (ICFG) and was reported to have little loss of precision with no loss of (100%) recall relative to static slicing based on the TSD model (i.e., the SDG). Later, the SEA was applied to static impact analysis shown more accurate than peer techniques [24] and capable of improving regression test selection and prioritization [14].

However, previous studies on the accuracy of SEA/SEB either exclusively targeted procedural programs [7], or focused on backward dependencies based on the SEB (against backward slicing on top of the SDG) only [18]. The remaining relevant studies addressed the accuracy of SEA-based forward dependencies, with some indeed using object-oriented programs and compared to forward slicing on the TSD model, yet the accuracy of such dependencies was assessed either not at the method level, but at class level only [25], or not relative to ground truth based on the TSD model, but those based on repository changes [13, 14] or programmer opinions [24], and only in the specific application context of impact analysis.

While forward dependence analysis is required by many dependence-based applications, including *static* impact analysis that the SEA/SEB has been mainly applied to, the accuracy of this abstraction with respect to the TSD model, for forward dependencies and object-oriented programs in particular, remains unknown. In addition, it has not yet been explored whether and, if possible, how such program dependence abstractions would improve *dynamic* analysis, especially hybrid ones that utilize both static dependence and execution data of programs, such as hybrid dynamic impact analysis [26, 27, 28].

In this paper, we present and study an alternative method-level dependence abstraction using a program representation called the method dependence graph (MDG). In comparison to the SDG-based TSD models which represent a program in terms of the data and control dependencies among all of its statements, an MDG serves also as a general graphical program representation, but models those dependencies at method level instead. The method-level dependencies could be simply obtained from a TSD model by lifting statements in the SDG up to corresponding

(enclosing) methods. Yet, our MDG model represents these dependencies directly with statement-level details within methods (i.e. intraprocedural dependencies) abstracted away and, more importantly, does so with much less computation than constructing the SDG would require. The MDG computes transitive interprocedural dependencies in a context-insensitive manner with flow sensitivity dismissed for heap-object-induced data dependencies too. Thus, it is more efficient than TSD models [6, 29]. On the other hand, this abstraction captures whole-program control and data dependencies, including those due to exception-driven control flows [30], thus it is more informative than coarser models like call graphs or ICFG. With the MDG, we attempt to not only address the above questions concerning the latest peer approach SEA/SEB, but also to attain a more cost-effective dependence abstraction over existing alternative options in general.

We implemented the MDG and applied it to both static and dynamic impact analysis for Java[1], which are all evaluated on seven non-trivial Java subject programs. We computed the accuracy of the MDG for approximating forward dependencies in general and the cost-effectiveness of its specific application in static impact analysis; we also compared the accuracy and efficiency of the MDG with respect to the TSD as ground truth against the SEA approach. To explore how the MDG abstraction can be applied to and benefit dynamic analysis, we developed on top of the MDG a variant of DIVER, the most cost-effective hybrid dynamic impact analysis in the literature [27], and compared its cost and effectiveness against the original DIVER.

Our results show that the MDG can approximate the TSD model with perfect recall (100%) and generally high precision (85-90% mostly) with great efficiency, at least for forward dependencies at the method level. We also found that the MDG appears to be a more cost-effective option than the SEA for the same purpose, according to its significantly higher precision with better overall efficiency. The study also reveals that, for the object-oriented programs we used at least, SEA can be much less precise for approximating forward dependencies at method level than previously reported at class level for object-oriented programs [25] and at method-level for procedural programs [7, 18]. The study also demonstrated that the MDG as a dependence abstraction model can significantly enhance the cost-effectiveness of both the dependence-based static and dynamic impact analysis techniques over the respective existing best alternatives. More broadly, the MDG as a general program abstraction approach could benefit any applications that are

[1]Source code, documentation, and study results are available at https://chapring.github.io/mdg

based on program dependencies at method level (e.g., testing and debugging) and that utilize the dependencies at this or even higher levels (e.g., refactoring and performance optimizations).

In summary, the contributions of this paper are as follows:

- An approach to abstracting program dependencies to method level, called the MDG, that can approximate traditional software dependencies more accurately than existing options, including dependencies due to exception-driven control flows (Section 3).
- An implementation of the MDG and two application analyses based on it, a static impact analysis and a hybrid dynamic impact analysis (Section 4.1).
- An extensive evaluation of the MDG that assesses its accuracy relative to TSD-based static slicing in approximating method-level forward dependencies, and that demonstrates its application and benefits for both static and dynamic analyses (Section 4).
- The first substantial empirical evidence on the accuracy of the SEA with respect to TSD-based static slices on object-oriented software, and that of the performance contrast between such dependence abstractions and the full TSD model those abstractions approximate (Section 5).

2. Motivation and Background

Our work was primarily motivated by improving the cost-effectiveness of forward dependence analysis that directly supports dependence-based impact analysis [12, 31], among many other software-evolution tasks [32]. The need for a better cost-effectiveness of impact analysis has been extensively investigated in previous studies (e.g., [17, 33, 34, 35, 36]) and recently stressed in [28]. This section highlights the problems with existing approaches that motivate our work as well as introduces background concepts and techniques necessary for understanding the rest of this paper.

An example program is also presented for illustrating purposes. In Figure 1, program E inputs two integers `a` and `b` in its entry method `M0`, manipulates them via `M1` and `M4` and prints the return values concatenated. `M4` updates the static variable `g` to be used by `M2` via a call to `M1`. `M3` and `M5`, invoked by `M1` and `M2`, include field accesses, conditionals, and arithmetics.

```
1  public class A {
2     static int g; public int d;
3     String M1(int f, int z) {
4        int x = f + z, y = 2, h = 1;
5        if (x > y)
6           M2(x, y);
7        int r = new B().M3(h, g);
8        String s = "M3val: " + r;
9        return s;}
10    void M2(int m, int n) {
11       int w = m - d;
12       if (w > 0)
13          n = g / w;
14       boolean b = C.M5(this);
15       System.out.print(b);}}
16 public class B {
17    static short t;
18    int M3(int a, int b) {
19       int j = 0;
20       t = -4;
21       if ( a < b )
22          j = b - a;
23       return j;}
```

```
24    static double M4() {
25       int x = A.g, i = 5;
26       try {
27          A.g = x / (i + t);
28          new A().M1(i, t);
29       } catch(Exception e) { }
30       return x;}}
31 public class C {
32    static boolean M5(A q) {
33       long y = q.d;
34       boolean b = B.t > y;
35       q.d = -2;
36       return b;}
37    public static void
38     M0(String[] args) {
39       int a = 0, b = 3;
40       A o = new A();
41       String s = o.M1(a, b);
42       double d = B.M4();
43       String u = s + d;
44       System.out.print(u);
45    }
46 }
```

Figure 1: The example program E used for illustration throughout this paper.

2.1. Impact Analysis

Impact analysis is a crucial step during software evolution [12, 32], for which a typical approach is to find the *impact set* (the set of potentially impacted program entities) of points of interest, such as those for change proposals, by analyzing program dependencies with respect to those points. Despite of many efforts invested [31], today's impact-analysis techniques still face many challenges, most of which can be reduced to the struggle between the cost and effectiveness of using the techniques or their results [16, 37, 38, 17, 8, 36].

In this context, a dynamic impact analysis DIVER [27, 28] was developed and shown to be much more precise than its previous alternatives with reasonable overheads. Given a program and its test inputs, this technique first builds a detailed statement-level dependence graph of the program, and then guides, using static dependence of that graph, the impact computation based on method execution traces generated from the test inputs. However, during the post-processing phase, intraprocedural dependencies kept in the graph carry excessive overheads as they cannot be pruned by the execution traces of method-level granularity (in essence, they are conservatively assumed to be all exercised due to the lack of statement-level dynamic data [28]). Therefore, for hybrid analysis using method-level execution data only, those intraprocedural dependencies can be abstracted away.

A few approaches devoted to abstracting program dependencies to method level exist [39, 29] which, however, are as heavyweight as or even more than the TSD model [6] that either do not scale to large programs or come with excessive costs. DIVER derives method-level dependencies from statement-level ones based on the TSD model, thus it also suffers from certain costs that could be avoided. Therefore, the overheads of DIVER would be reduced without losing precision, implying the increase in its overall cost-effectiveness, if we *directly* model method-level dependencies to capture only necessary information used by the dynamic analysis.

The applicability of dynamic impact analysis is constrained by the availability of program inputs (hence the dynamic data), though. When such inputs are not available, impact analysis would be performed with static approaches. In the current literature, the most cost-effective method-level static impact analysis we are aware of is based on the SEA relations among methods [13, 14]. Such analyses input a program and a *query* (a method for which impacts are queried), and outputs all methods that possibly statically execute after the query as the impact set. Yet, intuitively this approach can be very imprecise because of its highly conservative nature, as discussed and illustrated as follows.

2.2. The Static Execute After (SEA)

The static-execute-after (SEA) relation is defined as the union of *call*, *return-into*, and *sequentially follow* relations, all considered transitively [25]. For SEA computation, the analysis first creates the ICFG of the input program and then keeps entry and call-site nodes with the rest removed, followed by shrinking strongly connected components of the remaining graph into single nodes. Unfortunately, as a dependence abstraction of the TSD model, the SEA has not yet been fully evaluated against TSD-based ground truth for forward-dependence approximation, with widely used object-oriented software in particular (only differences between forward and backward dependence sets based on the SEA/SEB were reported in [18], yet still limited to procedural programs).

However, to inform developers about the reliability of results given by SEA-based impact analysis techniques, it is important to access SEA's accuracy in approximating forward dependencies on which the impact analysis is based. In addition, according to the definition of SEA, such impact analysis identifies dependencies among methods based on their connections via control flows only. Although data and control dependencies are realized through control flows at statement level, thus the approach is expected to be safe, ignoring the analysis

of those dependencies naturally results in false-positive dependencies. Yet, understanding the extent of such imprecision is still an unanswered but critical question.

To see how the SEA-based impact analysis works and its imprecision, consider the example program E of Figure 1 and method `M0` as the query. First, the query itself is trivially included in the impact set. Then, since `M0` calls `M1` and `M4`, and also transitively `M2`, `M3` (both via `M1`), and `M5` (further via `M2`), the impact set of `M0` is {`M0`, `M1`, `M4`, `M2`, `M3`, `M5`}. Similarly, for every other possible query, the impact set is constantly the entire program. However, these results are quite imprecise for this simple program: For example, none of `M1`, `M3`, and `M4` should be included in the impact set of `M5` because none of them is either data or control dependent on `M5`. We believe that properly incorporating data and control dependencies in a dependence abstraction would largely overcome such imprecision with acceptably additional yet still practical overhead.

2.3. Program Dependencies

Program dependencies are classified as control or data dependencies [1]. A statement s_1 is *control dependent* [5] on a statement s_2 if a branching decision taken at statement s_2 determines whether statement s_1 is necessarily executed. In Figure 1, for example, statement 22 is control dependent on statement 21 because the decision taken at 21 determines whether 22 executes or not. Another example is the set of statements 11, 12, 14, and 15 which are control dependent on statement 5, whose decision determines whether `M2` is called. These dependencies are *interprocedural* [40].

A statement s_1 is *data dependent* [41] on a statement s_2 if a variable v defined (written) at s_2 is used (read) at s_1 and there is a *definition-clear path* in the program for v (i.e., a path that does not re-define v) from s_2 to s_1. In Figure 1, for example, statement 36 is data dependent on statement 34 because 34 defines `b`, 36 uses `b`, and there is a path $\langle$34,35,36$\rangle$ that does not re-define `b`. In our work, we treat formal parameters as defined at the entry of each procedure (function or method) and also as data dependent on the corresponding actual parameter at each call site for that procedure. For example, the formal parameter `q` at statement 32 is a definition of `q` and also data dependent on the actual parameter `this` at 14.

3. The Method Dependence Graph

We first give an high-level description, followed by the definition, of the MDG, and then present in detail the algorithm for constructing the MDG on a given input program. We use both graph and code examples for illustration. This section

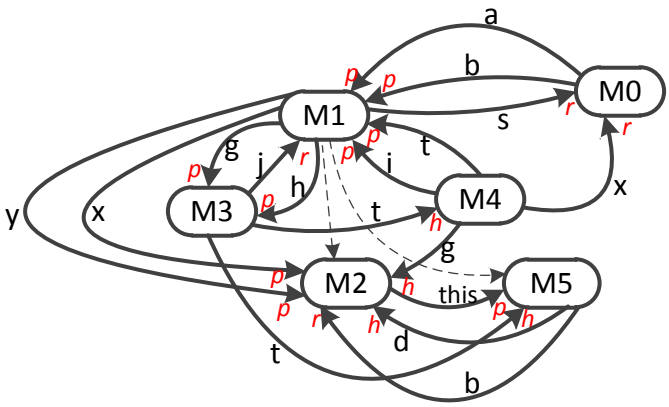

Figure 2: The method dependence graph (MDG) of program $E2$.

focuses on presenting the MDG technique itself as a generic program dependence model, with details on its use in impact analysis as an *example application* (mainly the implementation of two impact-analysis techniques based on this model) deferred to Section 4.1.

3.1. Overview

3.1.1. Definition

An MDG is a directed graph where each node uniquely represents a method and each edge a method-level data or control dependence. A method m' is data dependent on a method m if m defines a variable that m' might use, whereas a method m' is control dependent on a method m if a decision in m determines whether m' (or part of it) executes. In addition to traditional control dependencies due to ordinary control flows, the MDG also considers those caused by exception-driven control flows [30, 27]. As such, the MDG aims to *directly* represent data and control dependencies *among methods* as attempted in [39, 29] while ignoring unnecessary statement-level details. Figure 2 gives a quick look at an example MDG (for the example program of Figure 1), which is explained in 3.1.3.

To present the MDG, we refer to the specific target and source points at the boundary of a node, where edges enter and exit, as *incoming ports* (*IP*s) and *outgoing ports* (*OP*s), respectively. That is, an *IP* of a method m is an exact program point (statement) with respect to where m is dependent on other methods, and an *OP* of m is the point with respect to where other methods depend on m. Thus, the *interprocedural* dependencies among methods in a program are represented

by edges connecting *OP*s to *IP*s in the MDG of that program. We further refer to a dependence pointing to an *IP* of method m as an *incoming dependence* of m, and a dependence leaving an *OP* of m as an *outgoing dependence* of the method.

In contrast, *intraprocedural* dependencies are summarized by edges each directly connecting an *IP* to an *OP* inside an MDG node. We further refer to such edges as *summary edges* and, accordingly, the corresponding dependencies as *summary dependencies*. In other words, a summary dependence of a method m connects an *IP* to an *OP* of m, representing that the *IP* is reachable to the *OP* via at least one intraprocedural dependence chain inside m. As the MDG focuses on modeling method-level dependencies, it abstracts intraprocedural dependencies using summary dependencies inside methods, while maintaining incoming and outgoing dependencies across methods for the interprocedural dependencies among them. For brevity, we hereafter use edge and dependence interchangeably in the context of the MDG as a program representation.

More specifically, an MDG node n_m represents a method m, with a tuple that consists of three elements: the method identifier for m (e.g., method index), the list of *IP*s of m, and the list of *OP*s of m. An MDG edge from a method m to a method m' connects a specific *OP* of m to a specific *IP* of m', expressing either a method-level data or control dependence of m' on m. Therefore, the MDG of a program has the same number of nodes as that of the methods in the program, and the number of edges equal to that of the interprocedural edges in its corresponding (statement-level) dependence graph, plus the number of all summary edges. Maintaining the ports in each node and edges at port level is necessary for easily deriving more precise *transitive* method-level dependencies on the MDG than directly connecting the nodes with method-level edges only.

To facilitate the description of the MDG and the design of its application analyses (e.g., impact analysis based on the MDG), we also classify interprocedural data dependencies into three categories: *parameter* dependence connecting from actual parameters at a call site to formal parameters in the entry of the corresponding callee, *return* dependence from return statements to corresponding caller sites, and *heap* dependence from definitions to uses of heap variables (i.e., dynamically-allocated variables not passed or returned explicitly by methods).

3.1.2. Comparison

The MDG we present in this work shares some concepts and terms, such as incoming/outgoing port and dependencies, with the DIVER dependence graph [27], for which the interprocedural data dependence categorization was previously used too. Yet, there are substantial differences between these two representations. In-

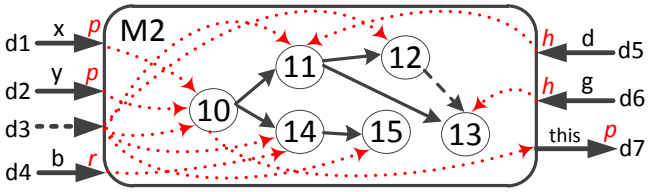

Figure 3: Statement-level dependencies in `M2` used to find *summary* dependencies directly connecting incoming dependencies to outgoing dependencies. These intraprocedural dependencies are *discarded* after the summary edges are created in the MDG.

stead of simply *renaming* vertices on interprocedural dependence edges of the dependence graph underlying DIVER to such terms after the detailed (fine-grained) dependence graph was constructed through a whole-program statement-level data and control flow analysis, we define the MDG based on these terms before constructing it, and use them to guide the construction process. In particular, we explicitly find the ports without computing interprocedural and intraprocedural dependencies beforehand, but instead first locating the incoming and outgoing ports, and then connecting them *among* methods; we compute intraprocedural dependencies for connecting incoming to outgoing ports *inside* methods and *discard them afterward*—they are not included in the MDG.

The DIVER dependence graph is in essence a context-insensitive system dependence graph with data dependencies between heap objects computed without flow sensitivity either, whereas the MDG focuses on modeling method-level dependencies thus abstracts away statement-level details within methods. As can be seen later in the MDG construction algorithms, the dependencies in the MDG are computed mainly through type-based port matching (rather than by solving flow equations iteratively to reach fixed points). Thus, the time that the MDG construction takes is less than the time that building the fine-grained dependence graph in DIVER would cost.

3.1.3. Illustration

As an illustration, the MDG of the example program E of Figure 1 is shown in Figure 2, where each node represents a method and each edge a dependence of the target node on the source one. Control dependencies (CDs) are depicted in dashed edges and data dependencies (DDs) are in solid edges. Each DD edge is annotated with the variable associated with the dependence and its arrow is labeled with the DD type (*p* for *parameter*, *r* for *return*, and *h* for *heap*). In this example, `M2` has

six incoming dependencies (five DDs and one CD), such as the DD labeled `g` with arrow *h* caused by the use of heap variable `g` defined in `M4`. Method `M2` has also an outgoing DD edge to `M5` because it calls `M5` with parameter `this`.

Figure 3 shows the original statement-level dependencies *within* `M2` (i.e., its program dependence graph (PDG) [5]) and the incoming and outgoing dependencies of that method, named *d1–d7* for convenience. Dotted (not dashed) edges indicate the connections between method- and statement-level dependencies. For M2 and every other method, a reachability analysis on these connections identifies the summary dependencies of outgoing dependencies on incoming dependencies. For `M2`, only *d1*, *d2*, and *d3* reach the outgoing dependence *d7*. Thus, the *summary dependencies* for `M2` are $\langle d1, d7\rangle$, $\langle d2, d7\rangle$ and $\langle d3, d7\rangle$. As mentioned earlier, the MDG does not keep the PDGs but these summary dependencies instead.

3.2. Construction of the MDG

Finding the dependencies in the MDG requires an *intraprocedural* analysis of the *statements* of each method. However, unlike statement-level graphs [6], only summaries of reaching definitions, reachable uses, and control dependencies for *call sites* and *exit nodes* are kept in memory after each intraprocedural analysis. For the CDs, we first compute *regular* CDs caused by branches, polymorphic calls, and intraprocedural exception control-flows. For the remaining *interprocedural* exception control-flows [40], we compute the CDs for the exception types not handled by the originating methods. These exception-induced CDs are detected using the same exception profiler as used in DIVER.

The MDG construction uses *intraprocedural* statement-level information for each method. This information is *discarded* after analyzing each method, once ports and summary dependencies are identified. Hence, the algorithm does not incur the time and space overheads of statement-level interprocedural analysis. Only necessary information for method-level dependencies is kept.

3.2.1. Identification of Ports

For a node (method) m, the *IP*s of m for DDs are the *uses* of variables v in m reachable from definitions of v in other methods or recursively in m. The *OP*s of m for DDs are the *definitions* in m that reach uses in other methods or recursively in m. The following are the cases in which ports for DD can be identified:

- For heap variables, including exceptions, whose definitions are *OP*s and whose uses are *IP*s

Algorithm 1 : BUILDMDG(program P, exception set *unhandled*)

```
1:  G := empty graph // start with empty MDG of P
2:  IP := OP := ∅   // maps of methods to incoming/outgoing ports
    // Step 1: find ports
3:  for each method m of P do
4:      FINDDDPORTS(m, IP, OP)
5:      FINDCDPORTS(m, IP, OP)
    // Step 2: connect ports
6:  for each method m of P do
7:      for each DD port z ∈ OP[m] do
8:          add {⟨z, z'⟩ | ∃m' s.t. z'∈IP[m'] ∧ data_dep(z,z')} to G
9:      MATCHINTERCDS(G, unhandled, m, IP, OP)
10:     pdg := GETPDG(m)
11:     for each port z ∈ IP[m] do
12:         add {⟨z, z'⟩ | z'∈OP[m] ∧ reaches(z,z',pdg)} to node G_m
13: return G
```

- For method calls, where actual parameters (at call sites) are *OP*s and formal parameters (at method entries) are *IP*s
- For method returns, where returned values at callees are *OP*s and returned values at caller sites are *IP*s

The CD *IP*s of a method m are denoted by special locations within m whose executions depend on external (or recursive) decisions such as calling m or returning to m with an unhandled exception. Those control-flow decisions are the *OP*s. Concretely, CD ports are identified for these cases:

- The entries of all methods that can be invoked are *IP*s. For a non-polymorphic call site (which can only call one method), every branch and CD *IP* that guards it is an *OP*. For a polymorphic call site (which has multiple target methods), the call site itself is an *OP* because it decides which method is called
- For an unhandled exception x thrown by a method m, the *entry points* of all blocks that can handle x (e.g., *catch* statements) at callers of m are *IP*s. The conditions that cause the exception to be thrown (i.e., the branches and CD *IP*s that guard its throwing or the instruction that conditionally throws it) are *OP*s

The cases listed for DD and CD ports set the rules for matching ports to determine the (interprocedural) DD and CD edges of the MDG—an *OP* can connect only to an *IP* for the same case. Thus, an MDG edge e from m to m' links an *OP* of m (the source of e) to a compatible *IP* of m' (the target of e) as per these cases.

An MDG node represents a method, its *IP*s, its *OP*s, and the *summary dependencies* that map *IP*s to *OP*s in that method. An *OP* p_o is summary-dependent on an *IP* p_i if there is a path from p_i to p_o in the (intraprocedural) statement dependence graph of the method [5]. With this information and the MDG edges, a client analysis such as dynamic dependence-based impact analysis (DDIA) [28, 11] can find which methods are impacted by a method m by traversing the MDG from m and all *OP*s of m conditioned to edges whose source *OP*s are summary-dependent on the *IP*s that are targets of traversed (impacted) edges.

3.2.2. Construction Algorithm

The main algorithm

Algorithm 1 describes the process for building an MDG. We use these helper notations: a caller (call) site *crs* (*cs*) is a tuple $\langle m, s \rangle$ where m is the caller (set of callees) and s the calling statement; *actual_params*(*cs*) is the actual parameter list of a call site *cs* and *formal_params*(m) the formal parameter list of m; *return_sites*(m) is the set of return statements in m and *return_type*(m) the return type of m; *D(rs)* is the definition of the return value in a return statement *rs*; *U(crs.s,rs)* is the use at a caller site *crs* of the value returned by a return statement *rs* in a method called by *crs*. We also denote a formal parameter f at the entry of m as the use *U(f,m)* and an actual parameter a in a call site *cs* as definition *D(a,cs)*.

The algorithm inputs program P and a set of unhandled exceptions and outputs the MDG of P. The exception set contains, by default and for static-analysis clients, all possible exceptions for safety; for dynamic-analysis clients, as mentioned earlier, the exception profiler identifies a potentially-smaller set for efficiency while staying safe, using the execution sets utilized by that client analysis. First in the algorithm, the DD and CD ports are identified for all methods of P via FINDDDPORTS and FINDCDPORTS, respectively. Next, the algorithm creates all DD edges, CD edges, and summary dependencies by connecting the ports that match (e.g., actual and formal parameters).

DD edges between methods are created in lines 7 and 8 by matching each DD *OP* z to each DD *IP* port z' that may be data dependent on z according to any of the three cases described earlier. Specifically, for safety and efficiency, all DDs are matched essentially based on the call graph without considering calling contexts

Algorithm 2 : FINDDDPORTS(m, *IP*, *OP*)

```
1:  for each call site cs in m do
2:      for each callee m' of cs do
3:          add {D(a, cs) | a ∈ actual_params(cs)} to OP[m]
4:          add {U(f, m') | f ∈ formal_params(m')} to IP[m']
5:  if return_type(m)≠void then
6:      add {D(rs) | rs ∈ return_sites(m)} to OP[m]
7:      for each caller site crs of m do
8:          add {U(crs.s,rs) | rs ∈ return_sites(m)} to IP[crs.m]
9:  for each heap variable definition hd in m do  add hd to OP[m]
10: for each heap variable use hu in m do  add hu to IP[m]
```

(i.e., ignoring context-sensitivity). CD edges are created via MATCHINTERCDS in line 9, which matches CD ports according to the rules for CDs listed earlier. CDs due to exceptions are included only for exceptions in the set *unhandled*.

Finally, the algorithm computes the *summary* dependencies within each method (lines 10–12). For each method m, given its PDG [5] (line 10) which contains all intraprocedural dependencies, the algorithm matches each *IP* with every *OP* that the *IP* can reach in that PDG. For each match, a summary edge $\langle z, z' \rangle$ is added to the node G_m for m in MDG G (line 12).

Collecting ports

The helper Algorithm 2 shows the details for FINDDDPORTS. For the input method m, the algorithm first traverses all call sites to find, for each call site and callee, the definition and use of actual and formal parameters, respectively (lines 1–4). Then, for methods that return values (line 5), the returned values are added, as pseudo-definitions, to the *OP*s of m (line 6) and the use of that value at each caller site are added to the *IP*s of the caller methods (lines 7–8). Finally, the algorithm finds and adds all definitions and uses of heap variables in m to the corresponding *OP* and *IP* sets.

The helper Algorithm 3, FINDCDPORTS, first identifies as *IP* the entry point of m. This point represents the decision to enter the method, which in a PDG is the *true* outcome of the *Start* node [5]. Then, using the control-dependence graph (CDG), lines 2–5 mark as *OP*s the decisions that guard single-target calls and unconditional throwers of unhandled exceptions. Those decisions can be branches, the entry of m (target of caller dependencies), and targets of callee dependencies (interprocedural exception catchers and calls to methods that might return abnormally [40]). Then, all multi-target call sites in m are added to the *OP*s

Algorithm 3 : FINDCDPORTS(m, *IP*, *OP*)

```
1:  add entry of m to IP[m]  // entry represents all CD targets for callers
2:  for each edge ⟨h, t⟩ in GETCDG(m) do
3:      if t is a single-target call site then {add h to OP[m]}
4:      if t unconditionally throws unhandled exception in m then
5:          add h to OP[m]
6:  for each multi-target call site cs in m do {add cs.s to OP[m]}
7:  for each statement s in m do
8:      if s catches interprocedural exception then {add s to IP[m]}
9:      if s conditionally throws exception unhandled in m then
10:         add s to OP[m]
```

of m (line 6). Lines 7–10 find the *IP*s that catch interprocedural exceptions and *OP*s that throw exceptions conditionally (e.g., null dereferences).

Let N be the largest number of nodes in the control flow graph, and F the largest number of formal parameters, of any method in program P. Also, let M be the number of methods defined in P. The largest numbers of return sites, call sites, and heap variables of any method are all bounded by $\mathcal{O}(N)$ and the largest number of caller sites of a single method is bounded by $\mathcal{O}(MN)$. Algorithm 2 takes $\mathcal{O}(NMF)$+$\mathcal{O}(N^2M)$+$\mathcal{O}(N)$ time (for lines 1–4, lines 5–8, and lines 9–10, respectively). Since generally $F \ll N$, the cost of this helper algorithm is bounded by $\mathcal{O}(N^2M)$. The three `for` loops in Algorithm 3 cost $\mathcal{O}(N^2)$, $\mathcal{O}(N)$, and $\mathcal{O}(N^2M)$, respectively, thus the algorithm takes $\mathcal{O}(N^2M)$ time too. Thus, the first `for` loop (*Step 1*) in Algorithm 1 costs $\mathcal{O}(N^2M^2)$ time; the cost of the second step (*Step 2*) is also bounded by $\mathcal{O}(N^2M^2)$ where the data dependence checking takes $\mathcal{O}(NM)$ while checking the reachability on the PDG only needs constant time with precomputed reachability results for all *IP*s stored in the *IP* entries per method. Constructing the PDG for any method, which subsumes the CDG of the same method, costs $\mathcal{O}(N^2)$ time [5]. Therefore, the total time complexity of the holistic MDG construction algorithm is (quite loosely) bounded by $\mathcal{O}(N^2M^2)$. For each method, the information for $\mathcal{O}(N)$ ports needs to be stored. The precomputed reachability results (boolean values) cost $\mathcal{O}(N^2M)$ space for all methods. Thus, the total space complexity is $\mathcal{O}(N^2M)$, also a loose upper bound. Practical time and space costs are expected to be much lower than these worst-case bounds.

4. Empirical Evaluation

We evaluated our technique as a dependence abstraction in general and its applications to both static and dynamic impact analysis techniques in particular. For that purpose, we performed two empirical studies. In the first study, we computed the precision and recall of forward dependence sets derived from the MDG against forward static slices, both at method level, and compared the same measures and efficiency against the SEA. In the second study, we built a dynamic impact analysis based on the MDG, called MADGER, and compared its precision and performance against DIVER.

Accordingly, we formulated the following four research questions:

- **RQ1** How accurately can the MDG and SEA approximate the full TSD model in terms of forward dependence analysis?
- **RQ2** Can the dependence abstractions (MDG and SEA) archive significantly better efficiency than the TSD model for forward dependence analysis?
- **RQ3** Are the MDG and the static impact analysis based on it more cost-effective than the SEA approach?
- **RQ4** Does the dependence abstraction (MDG) reduce the cost without losing effectiveness of a hybrid dynamic impact analysis such as DIVER?

The rest of this section presents our tool implementation, experiment setup, and threats to the validity of our evaluation that are shared by both studies. Details including experimental methodology and empirical results are given in Section 5 and Section 6, respectively.

4.1. Implementation

We implemented the MDG, the SEA, MADGER, and the method-level TSD-based forward static slicer all on top of our Java analysis and instrumentation toolkit DUA-FORENSICS [42], which is built on the Soot static analysis framework [43]. To compute control dependencies, including those due to exception-driven control flows, we used the *exceptional control flow graph* (ExCFG) [27, 44] offered in the Soot framework.

The ExCFG was also employed to create the interprocedural component control flow graph (ICCFG) [25, 7], on which the SEA was implemented using the *on-demand* algorithm presented in [18]. For static slicing, we used the forward static slicer as part of DUA-FORENSICS with results lifted to method level. Both

Table 1: STATISTICS OF EXPERIMENTAL SUBJECTS

Subject	Description	#LOC	#Methods
Schedule1	priority scheduler	290	24
NanoXML	XML parser	3,521	282
Ant	Java project builder	18,830	1,863
XML-security	encryption library	22,361	1,928
JMeter	performance gauge	35,547	3,054
JABA	bytecode analyzer	37,919	3,332
ArgoUML	UML modeling toolkit	102,400	8,856

the slicer and SEA implementations utilized the same call graph facilities given by Soot with rapid type analysis (RTA) applied. More details regarding the slicer, such as points-to analysis and library-call modeling, can be found in [42].

Finally, we implemented MADGER as a variant of DIVER with the statement-level dependence graph replaced with the MDG and, for impact computation, the reachability querying from an *IP* p_i to an *OP* p_o within a method m replaced with directly checking the existence of a summary edge $\langle p_i, p_o \rangle$ of m [27]. A static impact analysis based on the MDG was also implemented, which simply gives as the impact set the transitive closure on the MDG starting from the input query (more specifically, starting from each *OP* of the query, and then taking the union of all such closures). The SEA-based impact analysis produces as the impact set of a given query all methods that are in a SEA relation with that query.

4.2. Experiment Setup

We selected seven Java programs of diverse application domains and sizes for our evaluation. Table 1 lists the basic characteristics of these subject programs, including the number of non-blank non-comment lines of code (*#LOC*) and number of methods (*#Methods*) defined in each subject. The first five subjects were all obtained from the SIR repository [45], for which we picked the first version available of each. We received the JABA [46] program from its authors and downloaded a revision (r3121) of ArgoUML [47] from its SVN repository.

All our experiments were performed consistently on a a Linux workstation with a Quad-core Intel Core i5-2400 3.10GHz processor and 8GB DDR2 RAM.

4.3. Threats to Validity

The main *internal* threat to the validity of our results is the possibility of having implementation errors in the tools used for our study (the analyses based

on the MDG and SEA, and the forward static slicer). However, the underlying infrastructure, Soot and DUA-FORENSICS, have both been tested, improved, and matured for many years. To reduce errors in the code written on top of these frameworks, we manually checked and verified the correctness using both the example program, and relatively smaller subjects Schedule1 and NanoXML for randomly sampled queries. An additional *internal* threat may come from possible errors in our experimental and data-analysis scripts. To minimize this risk, we tested and debugged those scripts and checked their functionalities against the requirements of our experimental methodology.

Another *internal* threat is the risk of misguiding the experiments with inaccurate ground truth from the static slicer. However, this threat has been mitigated in several ways. First, the slicer, as part of DUA-FORENSICS, has been used and stabilized along with the entire framework over the years. Second, most of the core facilities that could affect the accuracy of this slicer were directly adopted or extended from the Soot framework, which is a widely used static-analysis platform by many researchers. Finally, since those core facilities are shared by our implementation of the slicer, the MDG, and the SEA, possible biases, if any, in the results derived from comparing among these tools have been greatly reduced.

The main *external* threat to the validity of out study is our selection of subject programs. This set of seven subjects does not necessarily represent all types of programs used in real-world scenarios. To address this threat, we picked our subjects such that they were as diverse as possible in size, application domain, coding style, and complexity. Also, the subjects we used in our study have been extensively used by many researchers before.

For results related to dynamic impact analysis, an additional *external* threat concerns about the test suites used in the second study (Study II) in that the test suites may not exercise all behaviors of the respective subjects. Also, many methods were not covered by the test suites, so we could not apply the techniques to those methods. Thus, our results were interpreted in light of the extent of the ability of those test suites to exercise their subjects. To minimize the effect of this threat, we chose subjects for which reasonably large and complete test suites were provided—at least from a functional point of view. Also, most of these subjects and test suites have been used by other researchers in their studies, and therefore they may be seen as good benchmarks.

Yet another *external* threat is that the forward slicing algorithm we used in our study may not be the optimal one in terms of precision and efficiency, thus using a more sophisticated slicer would possibly lead to different study results. Similarly, while the SEA algorithm we adopted is the latest one we are aware

of that processes one query at a time, which is justified for comparing per-query processing time costs between the two abstraction approaches, different efficiency contrasts may be obtained if comparing the total time of processing all possible queries of a program at once (using the batch SEA-computation algorithm in [18]).

The main *construct* threat lies in our experimental design. Without any additional knowledge, we gave the same weight to the forward dependence sets (impact sets) of every method (query). However, in practice, developers may find some methods more important than others, and thus the reported precision results might not represent the actual results that developers would experience. To address this potential concern, we adopted the same experimentation procedure when obtaining the dependence sets from the two approaches we compared.

Finally, a *conclusion* threat to validity is the appropriateness of our statistical analyses. To reduce this threat, we used a non-parametric hypothesis testing which makes no assumptions about the data distribution (e.g., normality). Another *conclusion* threat concerns, in dynamic impact analysis, the data points analyzed: We applied the statistical analyses only to methods for which impact sets could be queried (i.e., methods executed at least once). However, we computed the results of all possible queries and reported those of covered ones per subject for our comparative study. For static-analysis based results (e.g., the accuracy of the MDG and SEA), we also collected data points for all methods in each subject as queries, in order to avoid relevant biases.

5. Study I: Approximating Method-Level Forward Static Dependencies

This section presents the main study, which addresses the accuracy of the MDG against the SEA relative to the TSD model. Since impact sets computed by the static impact analysis based on the MDG and SEA are also the method-level forward dependence sets used by the accuracy study, we *simultaneously* evaluate the accuracy of the two abstraction models and the static impact analysis techniques based on them. We also study the efficiency of all these approaches. This study aims to answer the first three research questions (RQ1-RQ3).

5.1. Methodology

We applied the MDG- and SEA-based static impact analysis tools, and the method-level TSD-based forward static slicer, to each of the seven subjects. We collected the forward dependence set (i.e., the impact set or method-level forward slice) of every single method defined in each subject as a query (i.e., method-level slicing criterion) by running each of the three tools on that query separately.

To obtain the method-level forward slice of a query from the slicer, we computed the statement-level forward slice of every applicable statement-level slicing criterion, and then took the union of the enclosing methods of statements in those slices. We also collected the CPU time elapse as the querying cost per such query. Next, we calculated the following metrics.

First, we calculated the precision and recall of forward dependence set produced by the MDG and SEA for each query using the corresponding forward slice given by the static slicer as the ground truth: The precision metric measures the percentage of dependencies produced by the abstraction approaches that are true positives (i.e., included in the forward slice), while the recall measures the percentage of dependencies in the forward slice that are included in the dependence set produced by the abstraction approaches. We report the distribution of the entire set of data points for these two metrics per subject.

The ground truth for impact analysis results could also be obtained from the actual impacts with respect to concrete changes made to the program under analysis [36, 48]. However, for code-based predictive impact analysis techniques [31] like SEA, DIVER, the MDG-based static and dynamic impact analysis, and previous such approaches (e.g., [37, 17]), the concrete changes are not available—the analysis only takes the program to be changed and candidate change locations. Also, these techniques essentially report program entities that are dependent on the given change locations as the potential impacts, corresponding to what forward slicing would compute if taking the change locations as the slicing criteria. Thus, it is appropriate to evaluate these impact analysis techniques using the corresponding forward slices as ground truth.

Second, we computed the forward-dependence querying time costs of the MDG, the SEA, and the forward static slicer. We report the time costs of building the program representations (i.e., ICCFG and the MDG) used by the abstraction approaches. These two types of costs are calculated separately to give more detailed efficiency results that users may need for better planning their budgets: The times for abstracting program dependencies are one-time costs in the sense that for any queries (criteria) the abstract dependence models can be reused in the query processing phase; while the querying time is incurred per individual query.

Finally, we applied a non-parametric statistical test, the Wilcoxon signed rank test [49], to assess the statistical significance of mean difference in each of the above two metrics (the accuracy and efficiency measures) between the two abstraction approaches against the method-level static forward slicing. For the statistical test, we adopted a confidence level of 95%, and the null hypothesis is that there is no difference in the means between the compared techniques. Thus,

we regard the difference significant if the Wilcoxon p-value is less than 0.05. In addition, we compared the static-analysis (graph construction) time costs of the MDG against the SEA. We also report the significance over all subjects when applicable by combing the per-subject p-values using the Fisher method [50].

5.2. Results and Analysis

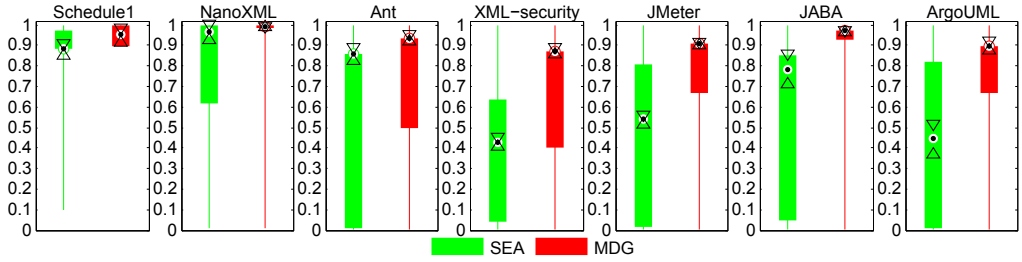

Figure 4: Precision of the MDG and SEA with the TSD-based static forward slices as the ground truth (constantly 100% recall for both approaches was obtained).

5.2.1. Precision (MDG and SEA versus TSD)

Figure 4 shows the precision results of the two approaches listed on the x axis, SEA and MDG, in approximating forward dependencies derived from the TSD model, where the y axis represents the precision. For each subject, a separate plot characterizes all the data points we analyzed, which consists of two boxplots each devoted to the result of one of the two approaches with that subject.

Each of the boxplots includes five components: the maximum (upper whisker), 75% quartile (top of middle bar), 25% quartile (bottom of middle bar), the minimum (lower whisker), and the central dot within each middle bar indicating the median. Surrounding each such dot is a pair of triangular marks that represent the comparison interval of that median. The comparison intervals within each plot together express the statistical significance of the differences in medians among the two groups in that plot: Their medians are significantly different at the 5% significance level if their intervals do not overlap.

The results indicate that the MDG can approximate the TSD-based forward dependencies with generally high precisions in most cases: For the majority of queries in all subjects, the precision was around 90%, according to the medians, and even low-end 25% of the queries had a precision between 45% (with XML-security the lowest) and 98% (with NanoXML the highest). For NanoXML and

JABA, the precision was over 95% for 75% of queries and as high as 90% for Schedule1. In these cases, we found many queries that share the same dependence sets, possibly due to the effects of dependence clusters [14]. The worst overall precision was seen by XML-security, for which the MDG gave a precision of no more than 85% for 75% of its 1,928 queries. Another subject that received mostly lower precisions than other subjects, except the worst-case XML-security, was Ant, where 25% of queries had results that were less than 55% precise.

While the MDG did not seem to have lower precisions for larger subjects—in fact, it performed almost as well with the two largest subjects as with the two smallest ones—the SEA did exhibit such trend, although not always. Except for the same worst case as can be seen by the MDG (with XML-security), all the other four largest subjects had generally much lower precisions from the SEA than from the MDG. Besides with Schedule1 and NanoXML, for which it was as almost precise as the MDG for the majority of queries, the SEA produced results of less than 5% precise for 25% of queries in other five larger subjects. For XML-security and ArgoUML, in particular, the precision did not even reach 50% for 50% of queries.

However, with both the MDG and SEA, there were cases in which the dependence abstraction missed almost all true-positive forward dependencies (precision close to zero), though much fewer seen by the MDG. We inspected random samples of such cases, where we found that mostly the results given by the abstraction approaches were large dependence sets with only one therein was true positive: the query itself. While the most possible reason was the conservativeness of the two approaches, the fact that the MDG has a lot less such bad cases than the SEA was more likely to lie in their different nature of technique: the MDG models both data and control dependencies, which tends to be less conservative than the SEA that considers more conservatively only control flows.

Supplementary to the boxplots, the left three columns of Table 2 gave an another statistics, the means of the precisions. Mainly due to the existence of bad cases discussed above, the means were dropped down considerably relative to the numbers seen in the distribution of all data points, for both approaches. Comparing between the MDG and SEA reveals that the SEA had an overall similar trend in the fluctuation of means across the seven subjects: where the MDG had relatively lower mean precisions, so did the SEA. Nonetheless, the p-values (in the fourth column) show strongly statistically significant differences in the means between the two approaches, further confirming the advantage of the MDG over the SEA on top of evidences from the significance in medians shown by the comparison intervals on the boxplots of Figure 4. To further understand the

advantage, we calculated the Cliff's deltas [51] as a non-parametric measure of the *effect size* of the difference between these two approaches in precision. With α=0.05 and a paired setting, the effect sizes for our benchmark subjects were generally beyond *medium* (¿=0.50, with most subjects) up to reasonably *large* (0.65, with JABA). Thus, given an arbitrary query, the MDG is expected to give a more precise forward dependence set than the SEA.

Over the total data points of all subjects, the MDG had a precision of 71% on average, significantly (p=0) higher than that of 49% archived by the SEA. Note that such means were largely skewed as discussed above. Nevertheless, for the SEA, in contrast to previous accuracy studies on it that reported very high precision, at class level [25] or at method level but with procedural programs (in C) considered only [7], our results suggest that object-oriented programs may contain much more method-level data and control dependencies that can not be accurately captured by control flows only, when compared to other cases (e.g., method-level forward dependencies on object-oriented programs) studied before.

Recall (MDG and SEA versus TSD). Given the conservative nature of both abstraction approaches and the consistency that all the studied techniques were implemented on a same source-code analysis infrastructure (call graph, points-to analysis, and data and control flow analysis facilities, etc.), we expected that both MDG and SEA are safe. Our results confirmed this hypothesis: for all the data points we collected, the *recall was constantly 100%* for both approximation models. Consequently, the precision numbers reported here can be readily translated to accuracy values (e.g., for a precision p, the F1 measure of accuracy is $2p/(1+p)$). Thus, we only show the precision in this paper for the evaluation on accuracy. Note that here we measured the recall with respect to method-level dependencies rather than at statement level as in the TSD. With the MDG abstracting away intraprocedural dependencies, it discards those details within each method. Therefore, compared to the TSD, the MDG misses all intraprocedural dependencies but potentially includes more (i.e., false-positive) interprocedural dependencies. However, as our empirical evidences corroborated, the MDG captures all dependencies at *method level* that the TSD model subsumes, which is the goal of our MDG abstraction.

Answer to RQ1: *Both the MDG and SEA can approximate the TSD-based forward dependencies safely; the MDG can also give high precision in most cases, while the SEA is significantly less precise in general.*

Table 2: Means of precision of the MDG versus SEA (the left six columns) and time costs of both abstraction approaches relative to forward static slicing (the rightmost four columns).

Subject	Mean precision			Abst. time (s)		Forward dependence querying time (ms): mean (stdev)			
	SEA	MDG	p-value	ICCFG	MDG	SEA	MDG	p-value	Slicing
Schedule1	0.81	0.94	2.35E-02	3	4	6 (3)	4 (2)	0.39E-02	124 (194)
NanoXML	0.77	0.88	2.04E-09	4	9	9 (12)	3 (4)	7.95074E-07	1,267 (3,095)
Ant	0.55	0.72	7.79E-79	17	130	64 (62)	45 (43)	1.09677E-08	34,896 (74,210)
XML-security	0.40	0.67	4.64E-83	22	77	50 (67)	43 (34)	4.47922E-16	24,092 (46,601)
JMeter	0.48	0.71	7.15E-168	5	116	325 (376)	104 (87)	5.12958E-26	37,950 (72,045)
JABA	0.58	0.84	1.96E-33	28	302	213 (201)	121 (221)	3.58023E-10	444,188 (801,631)
ArgoUML	0.45	0.70	9.74E-20	24	597	501 (559)	357 (298)	0.3085361	453,822 (1,082,813)
Overall	**0.49**	**0.71**	**0**	**14.7**	**176.4**	**183.8 (303.9)**	**85.1 (111.6)**	**1.71E-57**	**64,137.3 (294,000.9)**

5.2.2. Efficiency (MDG and SEA versus TSD)

The rest of Table 2 focuses on the efficiency of the two abstraction approaches and the TSD model, including the abstracting time (*Abst. time*) in seconds consumed by building the underlying graphs (*ICCFG* is used by the SEA), the time in *milliseconds* taken by the three techniques for querying forward dependence sets based on respective dependence models. For the latter, the table lists the means and standard deviations (*stdev*) of all data points for each subject. The p-values shown in the ninth column report the statistical significance of the Wilcoxon test on the mean differences in querying costs between the two approaches.

In terms of the abstracting time, since the MDG as a graph computes much finer-grained information than the ICCFG, the MDG approach costs always more than the SEA, as we expected. Yet, in almost all cases, this costs were quite affordable in absolute terms. The highest cost was seen with the largest subject ArgoUML. However, for a software of such scale, the time of less than 10 minutes seems to be reasonable. Moreover, this phase is required only once for all possible impact-set queries *and varying program input sets*, for the single program version the analysis works with at least—since the impact analysis is predictive, the impact set produced is with respect to the change location (i.e, the query method) rather than concrete changes at that location. Additionally, given that the MDG was implemented as a quick prototype for our study, there is large room for performance optimization in the implementation which is expected to incur less overheads. Yet, even with these potential optimizations and accommodations, the substantial increase in the static-analysis overhead with the MDG versus the SEA might still not be acceptable, especially for a large-scale software like ArgoUML. And whether it is acceptable in practice depends on the budget constraints and how the users balance the cost and effectiveness of the impact analysis, and also how they deal with the tradeoff between the efficiency in the static-analysis phase and that in the querying phase (as we discuss later below).

For a similar reason to that for the abstracting time difference, the MDG took more space than SEA. However, our results revealed that the space costs of both approaches were almost negligible with respect to today's storage configurations: the highest such cost was no more than 50M, seen by ArgoUML.

The querying-time costs to the right of the table show that the higher one-time costs of the MDG approach were quite paid off: compared to the SEA, it cost constantly less in terms of the mean querying time with also smaller variations. In addition, the p-values tell that, for all individual queries, the MDG was more efficient than the SEA with strong significance, in all cases but ArgoUML, possibly due to the much larger graph size for this subject. Following a similar procedure to the Cliff's delta computation used for RQ1, we obtained effect sizes ranging from .14 (with ArgoUML) to .46 (with Schedule1), confirming the advantage of the MDG over the SEA although the difference is generally not as large as seen in the precision comparison. In all, over all subjects, the mean querying cost on the MDG is 85ms, less than half of that incurred by the SEA (184ms).

Put together, in contrast to the SEA-based dependence abstraction, the MDG approach sacrifices the efficiency in the static analysis for faster answers to impact-set queries. In addition, the impact analysis based on the MDG offers much more accurate (i.e., significantly higher precision and perfect recall) impact sets than the SEA-based alternative. Also, it is reasonable to expect, for large programs in particular, that the potential time savings obtained from our approach versus the SEA as a result of inspecting much less false-positive impacts and getting the impact sets faster for each single query should well pay off and outweigh the increase in the static-analysis overhead in practical use scenarios of these techniques.

On the other hand, compared to the cost of the full TSD model (the last column), both abstraction models appeared to be much cheaper. For the two smallest subjects, querying the method-level forward dependencies was 100x faster than on the TSD model; for other subjects, the ratio was around 1000x. For example, the huge variation of the TSD querying costs suggests that in some cases, a single query can take as long as a few hours, as we experienced in experimentation. By average over all queries, the SEA costs 1.96% of the per-query time, in contrast to 0.59% by the MDG, relative to the TSD-based static forward slicing.

Answer to RQ2: *Both the MDG and SEA dependence abstraction models are reasonably efficient, and much (100x–1000x) cheaper than the TSD model for querying method-level forward dependencies.*

5.2.3. Cost-effectiveness for static impact analysis (MDG versus SEA)

From the above comparisons, it has been seen that the MDG as a TSD abstraction approach is generally much more precise than the SEA alternative, for approximating forward dependencies at least. The numbers in Table 2 also suggest that the MDG approach provides significantly faster dependence-set querying than the SEA too, at the cost of more but reasonably affordable static-analysis time. In addition, for the cases we studied, both approaches were confirmed to be safe, always giving perfect recall relative to the full TSD-based static slicing.

Note that the forward-dependence sets studied above can also be regarded as impact sets from the perspective of static impact analysis. Therefore, taken together, the advantages in the precision and querying efficiency of the MDG over the SEA imply that, in the presence of the higher dependence abstraction overhead which remains practical, the MDG-based static impact analysis is potentially more cost-effective than the SEA-based analysis.

As we noted before, the abstraction time of the MDG (and the ICCFG) is a one-time cost as the resulting dependence model can accommodate both different impact-set (i.e., forward dependence set) queries and different sets of program inputs (for the single program version analyzed by these impact-analysis techniques). In addition, while the time cost for computing a single query appears quite small compared to the model abstraction time, developers may need to query impact sets on demand and repeatedly for different queries [34]. Thus, the querying efficiency can also be a significant factor in practice.

It is plausible that the cost-effectiveness advantage of the MDG over SEA in the context of static impact analysis would not hold if developers using these techniques often perform very few queries on the analyzed program version before making changes to the version. In such cases, the higher cost of the MDG for the static analysis could dominate the total *analysis cost* incurred by the technique hence diminishes its merits over SEA. First, we believe that it is not common nor safe in practice to have a program changed so frequently without carefully assessing the potential impact of candidate changes, thus the presumptive cases are expected to be rare. Second, even in such rare cases, potential savings in the impact-inspection time due to the higher precision (effectiveness) of the MDG compared to the SEA-based approach may still ultimately render the MDG a more cost-effective option overall—the cost includes the human overhead for examining the resulting impact sets. In fact, the human overhead can readily dominate the total time spent on impact analysis on account of the slower human reasoning relative to the automated machine computation. Particularly, it is not

uncommon that inspecting some methods in the impact set that have complicated interaction with other methods can be quite time-consuming [8].

Answer to RQ3: *The MDG incurs expectedly larger one-time cost than the SEA, which is still affordable though; otherwise, the MDG tends to be significantly more cost-effective than the SEA for static impact analysis.*

6. Study II: Improving Hybrid Dynamic Impact Analysis

This section presents our secondary study, which addresses the application of static dependence abstraction in hybrid dynamic analysis using the dynamic impact analysis as an example. This study concerns the efficiency benefits of the MDG abstraction and seeks to answer the last research question (RQ4). We also examine the hypothesis that the MDG-based dynamic impact analysis (MADGER) gives as accurate impact sets as the DIVER technique. That is, we expect that MADGER improves querying performance over DIVER without losing accuracy hence gains in the overall cost-effectiveness.

6.1. Methodology

For this study, we applied MADGER and DIVER to the same seven subjects as used in the first study. For each subject, we collected the impact set given by the two analysis tools separately, using each single method of the subject as the query. We collected also the CPU time elapse as the querying cost for each query.

To obtain the method execution event trace per test case required by both tools, we used the test suite provided with each subject as shown in the first two columns of Table 3, where the third column gives the number of queries (methods) that executed in at least one per-test trace. Together with the total number of methods listed in Table 1, these numbers implicitly give the method-level coverage of the test suites. We then calculated the following metrics using the impact sets and querying costs for the covered queries only.

To measure the hypothetical improvement in impact-set querying performance of MADGER over DIVER, we first computed the ratio, as percentage, of the querying cost of MADGER to that of DIVER per query, in addition to the absolute numbers on the query time of each approach. We report the mean and standard deviation of such data points from all queries per subject. To verify the expected precision preservation of MADGER relative to DIVER, we compared their impact sets and calculated the set differences.

Finally, to examine whether the performance improvement is statistically significant, we performed the same statistical test on the querying costs of both tools.

6.2. Results and Analysis

Table 3: Mean impact-set querying time of the two dynamic impact analyses and the mean querying time ratios of MADGER to DIVER, with the statistical significance (p values) of mean differences in this cost.

Subjects	#Tests	#Queries	MADGER querying time (seconds)		DIVER querying time (seconds)		Querying time ratio: MADGER/DIVER (%)		Wilcoxon p-value (Effect size)
			mean	stdev	mean	stdev	mean	stdev	
Schedule1	2,650	20	8.3	2.8	14.3	6.3	58.94	5.19	1.65E-06
NanoXML	214	172	4.1	5.8	5.1	6.8	88.18	9.15	3.50E-02
Ant	112	607	1.9	4.8	2.4	5.5	71.60	28.64	1.88E-12
XML-security	92	632	4.6	5.7	7.0	9.1	71.64	26.02	1.27E-08
JMeter	79	732	1.3	4.0	1.9	4.2	81.92	29.54	4.80E-03
JABA	70	1,129	35.0	38.4	38.8	41.9	83.84	29.50	2.10E-02
ArgoUML	211	1,098	8.0	33.8	9.3	39.9	84.10	35.88	1.71E-05
Overall average			**12.7**	**31.2**	**14.8**	**33.4**	**80.50**	**30.32**	**1.08E-27**

6.2.1. Querying performance

The performance of impact computation of MADGER is compared to that of DIVER in Table 3, including the mean and standard deviations (*stdev*) of the querying time of both techniques (the fourth to seventh columns) and the ratios of MADGER over DIVER (the eighth and ninth columns), with the Wilcoxon p-values for the differences in the querying costs (the last column). The bottom row shows the measures for all data points over the seven subjects (not for those shown in the above rows).

This table shows that, overall, MADGER can considerably improve the querying performance of DIVER, by 12% (for NanoXML) to 41% (for Schedule1) on average. The generally small standard deviations with respect to corresponding means indicate that such performance gains were relatively steady for any individual queries. In addition, the difference in the mean querying time was all significant statistically, according to the p-values that were constantly lower than the α value of 0.05. In some cases, such as Ant and XML-security, the significance values were very strong (close to 0). We also measured the effect sizes as for RQ1 and RQ2 with the same test settings but here for the difference between DIVER and MADGER in the querying time. The results in the range of 0.62 (with ArgoUML) to 1 (with Ant) with a median of 0.83 confirmed that the MDG model is apparently more efficient than the fine-grained dependence representation used in DIVER for the DDIA technique. Over all subjects, the performance gain was about 20% by MADGER in contrast to DIVER.

Individually, the largest querying-cost reduction, of over 40%, was seen by Schedule1, which is the smallest subject but has the largest number of test cases. This subject has also the smallest variation of such reduction compared to other subjects. Second to Schedule1, XML-security and Ant also had large improvements, close to 30% by average. In contrast, the three largest subjects, JMeter, JABA, and ArgoUML, had noticeably smaller improvements.

For most of these subjects, it appears that the performance gains were inversely correlated with the subject sizes: MADGER tends to archive smaller improvements against DIVER for larger programs. The only exception is NanoXML, which received the lowest querying-cost reduction among all seven subjects. This observation is not surprising, though, given the technical difference between DIVER and MADGER. As we confirmed from manual inspection, the main reason underneath this observation is that as program sizes grow, the programs tend to have higher degree of logic complexity in their interface among methods, which potentially results in larger numbers of interprocedural dependencies than intraprocedural ones in their DIVER dependence graphs—-MADGER, however, right focuses on reducing the number of intraprocedural dependencies through the dependence abstraction in the MDG. For example, JABA has very tight inter-function couplings due to its design choice of trying to separate the core functionality into many small-sized functions to facilitate internal code reuse. Similar code structure was found in ArgoUML as well, in which the layered architecture reflected by a large number of interconnected modules (packages) and components (classes) leads to dense dependencies among methods of relatively small sizes. Other relevant factors may include execution trace length, test input size, and subject nature [38, 27, 28]. We also manually checked the exceptional case NanoXML, and found that this subject, although relatively small in source size, has relatively high inter-method couplings and simple logic in most individual methods. Consequently, interprocedural dependencies that have to be kept in the MDG largely outnumber intraprocedural dependencies that the MDG tries to simplify.

On the other hand, the absolute numbers of querying time of MADGER versus DIVER suggest that the efficiency improvements obtained through the dependence abstraction via the MDG is not trivial: by average, querying on the MDG is faster than on the DIVER dependence graph by a few seconds *for each single query*. Furthermore, the constantly larger standard deviations of the means of DIVER over MADGER imply that the efficiency improvements are even higher in many cases—such improvements especially matter for those queries incurring the highest time overheads. Over all subjects, the dependence abstraction of the MDG reduced the mean impact-set querying time by over two seconds. When querying multiple

queries, the aggregate reduction to be achieved by the MDG tends to be even more significant relative to the SEA.

6.2.2. Other Costs

Additional costs involved in the total cost of DIVER include the static-analysis time, runtime overhead, and space costs [27]. However, in terms of overhead, the static-analysis phase incurs a one-time cost, thus it is usually a less critical part in the overall analysis process, as discussed in Section 5.2. As we expected, the time for constructing the MDG in MADGER was in fact always considerably (5–15%) smaller than that for constructing the DIVER dependence graph (as explained in Section 3.1.1); other parts of the static-analysis phase are the same between the two. We omit those data here for the above mentioned reasons. The runtime phase is the same between MADGER and DIVER, where the static dependence information is not used. The space costs for both tools were quite small, with the largest of such costs being less than 50MB as mentioned before.

Effectiveness (impact-set accuracy) We also compared each pair of impact sets produced by MADGER and DIVER to examine our hypothesis that the MDG did not introduce imprecision because of the dependence abstraction. According to our results on impact-set differences between the two tools, we confirmed that for every query MADGER produces the same impact set as DIVER. This was expected because the MDG captures the same essential information that this analysis requires as the DIVER dependence graph provides.

In all, MADGER appeared to be 20% more efficient than DIVER without losing precision or recall. Since the only difference between the two tools lies in the static dependence model underneath, our results suggest that the MDG can enable a hybrid dynamic impact analysis like DIVER to increase its cost-effectiveness significantly. Note that while such improvement may seem trivial for queries that the analysis technique can answer sufficiently fast, for those that take substantially longer time [28], a 20% reduction on average could imply quite useful savings in practice, given the restricted budget for impact analysis of developers [33].

While other optimizations exist, such as parallelizing the querying phase of the impact analysis for multiple queries, those optimizations are orthogonal to the improvement in the MDG. In cases where the querying cost remains challenging (for programs with very long execution traces, for instance), efforts focusing on parts other than the static dependence model in the overall analysis process can be invested in collaboration with the dependence abstraction via the MDG.

Answer to RQ4: *The MDG dependence abstraction can significantly improve the efficiency of a hybrid dynamic impact analysis like* DIVER *while without*

losing effectiveness (precision), hence gain in the overall cost-effectiveness of the dynamic impact analysis as a whole.

7. Related Work

We mainly discuss three categories of previous work related to ours: program dependence abstraction, static impact analysis, and dynamic impact analysis.

7.1. Program Dependence Abstraction

Most existing dependence-abstraction approaches were designed for specific applications, and mostly either do not directly model or not subsume complete method-level (data and control) dependencies. For instance, the *program summary graph* [19] was originally developed to speed up interprocedural data-flow analysis, which is similar to some data dependence abstraction parts of the MDG, but it is built based on program call structure only and, thus, it can be not only imprecise but also unsafe from the perspective of approximating a full TSD-based program dependence model.

Particularly targeting impact analysis, several abstract dependence models were proposed, including the *object-oriented class-member dependence graph* [20], the *lattice of class and method dependence* [21], and the *influence graph* [22]. These abstractions consider only partial dependencies: The former two only capture structural dependencies among classes, methods, and class fields that are directly derived from objected-oriented features, such as class-method memberships and class inheritance, and method-call relations; the last one captures only data dependencies among methods in an overly conservative manner while ignoring the analysis of intraprocedural dependencies, which has been shown highly imprecise (precision close to a transitive closure on the ICFG) [22]. The RWSets tool in [52] abstracts data dependencies via field reads and writes but ignores control and other data dependencies.

A few other approaches explicitly attempted to model method-level dependencies. One example is the *abstract system dependence graph* (ASDG) [29], which is built by first creating the entire SDG and then simplifying statement-level edges thereof. In [39], an extended TSD model is described to generalize the definitions of interprocedural dependencies, directly at both statement and procedure levels. Since these models were straightly derived from or developed atop the (SDG) [6], they are at least as heavyweight as the underlying TSD model itself.

The SEA/SEB abstraction [7] to which we compared the MDG is developed on the simplified ICFG (called ICCFG) in order to capture method execution orders

statically, which is motivated by the dynamic version of such orders proposed in [17]. Developed also for interprocedural data-flow algorithms, the *program super graph* [53] connects per-procedure control-flow graphs with call and return edges, similar to the ICFG but enclosing calling contexts as well. The context-sensitive CFG in [54] is proposed to visualize CFGs for program comprehension, which also simplifies each intraprocedural CFG as for the SEA/SEB.

In contrast, the MDG we proposed directly models method-level dependencies that explicitly include both data and control dependencies. However, compared to the TSD model, the MDG avoids expensive interprocedural data-flow analysis with context-sensitivity ignored as well, which makes it conservative yet enables it to be relatively lightweight. The summary edges in the MDG are also different from those of the same name used in the SDG and ASDG: Those edges were used to help represent calling contexts in the SDG and transitive data-flow across procedures in the ASDG; we use such edges to abstract reachability from incoming to outgoing dependencies within each method.

When developing the MDG, we reused some terms, such as the port and data-dependence classification, from DIVER [27]. Also, in terms of the graph representation, the MDG can be viewed as a more compact version of the DIVER dependence graph. However, in contrast to the MDG, that dependence graph is essentially a context-insensitive version of the SDG. In addition, unlike the MDG which is intended for a general lightweight TSD approximation (although initially motivated by our work on impact analysis), the dependence graph used by DIVER targets a specific application of static dependencies in hybrid dynamic impact analysis. In this sense, the MDG *generalizes* the DIVER dependence graph for a broader range of applications.

To address the scalability and usability challenges of the TSD model, approaches such as thin slicing [55] seek from a different perspective for simplifying the traditional dependence model by taking the concept and definition of dependencies of a constrained sense to prune less relevant dependencies, as was also examined in dynamic dependence analysis area [56]. The MDG shares similar goals with such techniques but attempts to directly reduce the TSD model to support dependence analysis of higher granularity rather than adapting the traditional dependence definitions or concepts.

7.2. Static Impact Analysis

Static impact analysis [57] provides impact sets for all possible program inputs. At method level, a main approach to such analysis is to find methods that are directly or transitively dependent on the given query. In comparison to the

SEA-based impact analysis that requires a reachability algorithm [13, 24], a static impact analysis based on the MDG simply computes the transitive closure from the query. The MDG-based impact analysis also gives more information regarding *how*, in addition to whether, impacts propagate across methods (through the ports and edges among them), thus it tends to better support impact inspection and understanding, than the SEA-based approach.

Static slicing has been directly used for static impact analysis, but it was shown to have challenges from overly large results and/or prohibitive costs [8]. Many other static approaches exist, which utilize various types of program information, such as code structure and version repository [31, 24]. Our static impact analysis based on the MDG utilizes method-level dependencies to offer an efficient approach with a precision comparable to static slicing. Others static approaches to impact analysis exploit concept lattice [58], or combine it with call graph [59], potentially providing an alternative to our technique based on the MDG.

7.3. Dynamic Impact Analysis

In contrast to static approaches, dynamic impact analysis uses concrete program execution sets to provide smaller impact sets specific to the program inputs utilized by the analysis [31, 37, 38]. For instance, EAS [17] identifies as impacted methods that execute after the input query, like the SEA/SEB approach, but leverages runtime data to improve the precision of results. We also utilized the execute-after relations as by EAS, but prunes false-positive impacts to gain even higher precision using a static dependence graph.

The dynamic impact analysis MADGER based on the MDG is an optimized version of DIVER [27]. By summarizing intraprocedural dependencies, MADGER avoids unnecessary overheads by carrying those dependencies in the dependence graph used by DIVER while losing no necessary information required for the analysis. Thus, compared to DIVER, MADGER archives higher impact-set querying performance without sacrificing precision, akin to the EAS optimization over its predecessor PATHIMPACT [16]. Previously, a number of other dynamic impact analysis techniques has been proposed, trying to continue improving the precision of EAS but no significant improvement was archived (e.g., [22, 60]).

Note that, with both static and dynamic approaches, a large body of other impact-analysis techniques has been developed but is *descriptive* [12, 31], such as SIEVE [61], CHIANTI [62], and the impact analysis based on static slicing and symbolic execution [63]. These approaches require prior knowledge about actual changes made across two program versions. In contrast, the impact analysis we focused on in this paper is *predictive*, which inputs a single program version

without knowing the actual changes, thus it gives prediction of possible impacts based on information from the single version of the program rather than describing the impacts of those known changes.

8. Conclusions and Future Work

Despite of a number of dependence abstractions proposed to approximate the fine-grained and heavyweight TSD model, only few of them intended for a safe and efficient general approximation. A recent one of such abstractions, the SEA/SEB, has been developed, yet it remains unclear how accurately this approach can approximate forward dependencies for object-oriented software. Also, our intuition and initial application of the SEA/SEB suggest that it may not be sufficiently accurate for that approximation.

Motivated by our work on impact analysis, we developed an alternative dependence abstraction, the MDG, which directly models method-level dependencies while giving more information than the SEA/SEB. For forward dependence approximation, we evaluated the accuracy and efficiency of the MDG against the SEA using fine-grained static forward slices uplifted to method level as the ground truth. We also implemented both a static and a dynamic impact analysis based on the MDG and evaluated their cost and effectiveness against the SEA-based alternative and one of the latest dynamic impact analysis techniques DIVER, respectively. We showed that the MDG is safe and highly precise, not only relative to the TSD but also strongly significantly more precise than the SEA, and that the MDG can improve the cost-effectiveness of both types of impact analysis with strong statistical significance.

There are considerable potentials of the MDG for other dependence-based applications, such as program comprehension, testing, and fault cause-effect understanding, which we plan to explore next. While our study results demonstrated that the MDG can be a better option for the TSD approximation, this paper is constrained to show that advantage for forward dependencies and at method level only. Thus, future study may consider addressing backward dependencies and at other levels of granularity.

Finally, while our work examines dependence abstractions relative to the TSD model, as previous work revealed and studied [25, 29], hidden dependencies that cannot be captured by the TSD model also exist, especially in object-oriented software, and can be discovered (partially) by approaches based on the SEA [25]. It would also be of interest to investigate in that regard using the MDG, especially in contrast to the SEA-based approaches.

Acknowledgments

This work was partially supported by ONR Award N000141410037 to the University of Notre Dame and faculty startup fund from Washington State University to the first author.

References

[1] A. Podgurski, L. Clarke, A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance, IEEE Transactions on Software Engineering 16 (9) (1990) 965–979.

[2] S. Bates, S. Horwitz, Incremental program testing using program dependence graphs, in: Proc. of Symp. on Principles of Program Lang., 384–396, 1993.

[3] R. Santelices, M. J. Harrold, Exploiting Program Dependencies for Scalable Multiple-path Symbolic Execution, in: Proceedings of ACM International Symposium on Software Testing and Analysis, 195–206, 2010.

[4] G. K. Baah, A. Podgurski, M. J. Harrold, The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis, IEEE Transactions on Software Engineering 36 (4) (2010) 528–545.

[5] J. Ferrante, K. Ottenstein, J. Warren, The program dependence graph and its use in optimization, ACM Trans. on Prog. Lang. and Systems, 9(3):319-349 .

[6] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Trans. on Prog. Lang. and Systems 12 (1) (1990) 26–60.

[7] J. Jász, Á. Beszédes, T. Gyimóthy, V. Rajlich, Static execute after/before as a replacement of traditional software dependencies, in: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, 137–146, 2008.

[8] M. Acharya, B. Robinson, Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems, in: Proceedings of IEEE/ACM International Conference on Software Engineering, Software Engineering in Practice Track, 746–765, 2011.

[9] D. Jackson, M. Rinard, Software Analysis: A Roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, 133–145, 2000.

[10] D. Binkley, Source code analysis: A road map, in: 2007 Future of Software Engineering, 104–119, 2007.

[11] H. Cai, R. Santelices, D. Thain, DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness, ACM Transactions on Software Engineering and Methodology (TOSEM) 25 (2) (2016) 18.

[12] S. A. Bohner, R. S. Arnold, An introduction to software change impact analysis, Software Change Impact Analysis, IEEE Comp. Soc. Press, pp. 1–26, 1996.

[13] J. Jász, L. Schrettner, Á. Beszédes, C. Osztrogonác, T. Gyimóthy, Impact analysis using Static Execute After in WebKit, in: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, IEEE, 95–104, 2012.

[14] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, T. Gyimóthy, Impact analysis in the presence of dependence clusters using Static Execute After in WebKit, Journal of Software: Evolution and Process 26 (6) (2014) 569–588.

[15] X. Ren, O. C. Chesley, B. G. Ryder, Identifying failure causes in java programs: An application of change impact analysis, Software Engineering, IEEE Transactions on 32 (9) (2006) 718–732.

[16] J. Law, G. Rothermel, Whole program Path-Based dynamic impact analysis, in: Proc. of Int'l Conf. on Softw. Eng., 308–318, 2003.

[17] T. Apiwattanapong, A. Orso, M. J. Harrold, Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences, in: Proc. of Int'l Conf. on Softw. Eng., 432–441, 2005.

[18] J. Jász, Static execute after algorithms as alternatives for impact analysis, Electrical Engineering 52 (3-4) (2010) 163–176.

[19] D. Callahan, The Program Summary Graph and Flow-sensitive Interprocedual Data Flow Analysis, in: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, 47–56, 1988.

[20] X. Sun, B. Li, C. Tao, W. Wen, S. Zhang, Change impact analysis based on a taxonomy of change types, in: IEEE Computer Software and Applications Conference, 373–382, 2010.

[21] X. Sun, B. Li, S. Zhang, C. Tao, X. Chen, W. Wen, Using lattice of class and method dependence for change impact analysis of object oriented programs, in: ACM Symposium on Applied Computing, 1439–1444, 2011.

[22] B. Breech, M. Tegtmeyer, L. Pollock, Integrating Influence Mechanisms into Impact Analysis for Increased Precision, in: Int'l Conf. on Softw. Maint., 55–65, 2006.

[23] S. Mancoridis, B. S. Mitchell, Y. Chen, E. R. Gansner, Bunch: A clustering tool for the recovery and maintenance of software system structures, in: Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on, 50–59, 1999.

[24] G. Tóth, P. Hegedűs, Á. Beszédes, T. Gyimóthy, J. Jász, Comparison of different impact analysis methods and programmer's opinion: an empirical study, in: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, ACM, 109–118, 2010.

[25] A. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, V. Rajlich, Computation of static execute after relation with applications to software maintenance, in: Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, IEEE, 295–304, 2007.

[26] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, D. D. S. Guerrero, The hybrid technique for object-oriented software change impact analysis, in: Euro. Conf. on Software Maintenance and Reengineering, 252–255, 2010.

[27] H. Cai, R. Santelices, DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning, in: Proceedings of International Conference on Automated Software Engineering, 343–348, 2014.

[28] H. Cai, R. Santelices, A framework for cost-effective dependence-based dynamic impact analysis, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 231–240, 2015.

[29] Z. Yu, V. Rajlich, Hidden dependencies in program comprehension and change propagation, in: Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on, 293–299, 2001.

[30] S. Sinha, M. J. Harrold, Analysis and testing of programs with exception handling constructs, Software Engineering, IEEE Transactions on 26 (9) (2000) 849–871.

[31] B. Li, X. Sun, H. Leung, S. Zhang, A survey of code-based change impact analysis techniques, Software Testing, Verification and Reliability 23 (2013) 613–646, doi:10.1002/stvr.1475.

[32] V. Rajlich, Software Evolution and Maintenance, in: Proceedings of the Conference on the Future of Software Engineering, ISBN 978-1-4503-2865-4, 133–144, 2014.

[33] P. Rovegard, L. Angelis, C. Wohlin, An empirical study on views of importance of change impact analysis issues, Software Engineering, IEEE Transactions on 34 (4) (2008) 516–530.

[34] C. R. de Souza, D. F. Redmiles, An empirical study of software developers' management of dependencies and changes, in: Proceedings of the 30th international conference on Software engineering, ACM, 241–250, 2008.

[35] M. Acharya, B. Robinson, Practical change impact analysis based on static program slicing for industrial software systems, in: Proceedings of the 33rd international conference on software engineering, ACM, 746–755, 2011.

[36] H. Cai, R. Santelices, T. Xu, Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis, in: Proceedings of International Conference on Software Security and Reliability, 48–57, 2014.

[37] A. Orso, T. Apiwattanapong, M. J. Harrold, Leveraging field data for impact analysis and regression testing, in: Proc. of 9th European Softw. Eng. Conf. and 10th ACM SIGSOFT Symp. on the Foundations of Softw. Eng., Helsinki, Finland, 128–137, 2003.

[38] A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, M. J. Harrold, An Empirical Comparison of Dynamic Impact Analysis Algorithms, in: Proc. of 26th IEEE and ACM SIGSOFT Int'l Conf. on Softw. Eng. (ICSE 2004), Edinburgh, Scotland, 491–500, 2004.

[39] J. P. Loyall, S. A. Mathisen, Using dependence analysis to support the software maintenance process, in: Software Maintenance, 1993. CSM-93, Proceedings., Conference on, IEEE, 282–291, 1993.

[40] S. Sinha, M. J. Harrold, G. Rothermel, Interprocedural control dependence, ACM Trans. Softw. Eng. Method. 10 (2) (2001) 209–254, ISSN 1049-331X, doi:http://doi.acm.org/10.1145/367008.367022.

[41] A. V. Aho, M. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques and Tools, Prentice Hall, 2006.

[42] R. Santelices, Y. Zhang, H. Cai, S. Jiang, DUA-Forensics: A Fine-Grained Dependence Analysis and Instrumentation Framework Based on Soot, in: Proceeding of ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis, 13–18, 2013.

[43] P. Lam, E. Bodden, O. Lhoták, L. Hendren, Soot - a Java Bytecode Optimization Framework, in: Cetus Users and Compiler Infrastructure Workshop, 2011.

[44] H. Cai, R. Santelices, TracerJD: Generic trace-based dynamic dependence analysis with fine-grained logging, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 489–493, 2015.

[45] H. Do, S. Elbaum, G. Rothermel, Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact, Empirical Software Engineering 10 (4) (2005) 405–435.

[46] A. R. Group, Java Architecture for Bytecode Analysis, `http://gamma.cc.gatech.edu/jaba.html`, [Online; accessed 03-Mar-2014], 2005.

[47] tigris.org, The ArgoUML Project, `http://argouml.tigris.org/`, [Online; accessed 03-Mar-2014], 2003.

[48] H. Cai, R. Santelices, A comprehensive study of the predictive accuracy of dynamic change-impact analysis, Journal of Systems and Software 103 (2015) 248–265.

[49] R. E. Walpole, R. H. Myers, S. L. Myers, K. E. Ye, Probability and Statistics for Engineers and Scientists, Prentice Hall, ISBN 978–0321629111, 2011.

[50] F. Mosteller, R. A. Fisher, Questions and Answers, The American Statistician 2 (5) (1948) pp. 30–31.

[51] N. Cliff, Ordinal methods for behavioral data analysis, Psychology Press, 1996.

[52] M. Emami, A practical interprocedural alias analysis for an optimizing/parallelizing C compiler, Master's thesis, McGill University, 1993.

[53] E. M. Myers, A precise inter-procedural data flow algorithm, in: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 219–230, 1981.

[54] J.-C. Ng, Context-sensitive control flow graph, Master's thesis, Iowa State University, 2004.

[55] M. Sridharan, S. J. Fink, R. Bodik, Thin slicing, in: Proceedings of ACM Conference on Programming Language Design and Implementation, 112–122, 2007.

[56] X. Zhang, N. Gupta, R. Gupta, Pruning Dynamic Slices with Confidence, in: Proceedings of ACM Conference on Programming Language Design and Implementation, 169–180, 2006.

[57] X. Sun, B. Li, H. Leung, B. Li, J. Zhu, Static change impact analysis techniques: A comparative study, Journal of Systems and Software 109 (2015) 137–149.

[58] P. Tonella, Using a concept lattice of decomposition slices for program understanding and impact analysis, Software Engineering, IEEE Transactions on 29 (6) (2003) 495–509.

[59] B. Li, X. Sun, H. Leung, Combining concept lattice with call graph for impact analysis, Advances in Engineering Software 53 (2012) 1–13.

[60] L. Huang, Y.-T. Song, A Dynamic Impact Analysis Approach for Object-Oriented Programs, Advanced Software Engineering and Its Applications 0 (2008) 217–220.

[61] M. K. Ramanathan, A. Grama, S. Jagannathan, Sieve: A Tool for Automatically Detecting Variations Across Program Versions, in: Proceedings of

IEEE/ACM International Conference on Automated Software Engineering, 241–252, 2006.

[62] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of java programs, in: Proc. of ACM Conf. on Obj. Oriented Prog. Syst., Lang., and Appl., 432–448, 2004.

[63] N. Rungta, S. Person, J. Branchaud, A change impact analysis to characterize evolving program behaviors, in: IEEE International Conference on Software Maintenance, 109–118, 2012.