Abstracting Program Dependencies using the Method Dependence Graph

Haipeng Cai and Raul Santelices
University of Notre Dame, Indiana, USA
email: {hcai|rsanteli}@nd.edu

***Abstract*—While empowering a wide range of software engineering tasks, the traditional fine-grained software dependence (TSD) model can face great scalability challenges that hinder its applications. Many dependence abstraction approaches have been proposed, yet most of them either target very specific clients or model partial dependencies only, while others have not been fully evaluated for their accuracy with respect to the TSD model, especially in approximating forward dependencies on object-oriented programs. To fill this gap, we present a new dependence abstraction called the method dependence graph (MDG) that approximates the TSD model at method level, and compare it against a recent TSD abstraction, called the Static-Exectue-After (SEA), concerning forward-dependence approximation. We also evaluate the cost-effectiveness of both approaches in the application context of impact analysis. Our results show that the MDG can approximate TSD safely, for method-level forward dependence at least, with little loss of precision yet huge gain in efficiency; and for the same purpose, while both are safe, the MDG can achieve significantly higher precision than SEA at practical costs.**

I. Introduction

Analyzing dependencies among program entities underlies a wide range of software analysis and testing techniques [1]. While traditional approaches to dependence analysis offer fine-grained results [2], they can face severe scalability challenges, especially with modern software of growing sizes and/or increasing complexity [3], [4]. On the other hand, for many applications where results of coarser granularity suffice, computing the finest-grained dependencies tends to be excessive, ending up with low overall cost-effectiveness. One example is impact analysis [5], where results are commonly given at method level [6]–[8], whereas statement-level results can be too large to fully utilize [4]. In other contexts such as program understanding, method-level results are also more practical to explore than are those of the finest granularity.

Driven by varying needs, different approaches have been explored to abstract program dependencies to coarser levels [9]–[11]. While these abstraction models have been shown to be useful for their particular client analyses, they either capture only partial dependencies among methods [10] or dependencies among components at the levels of granularity (e.g., class or even file level) which can be overly coarse for most tasks. More critically, none of such approaches were designed or fully evaluated as a general program dependence abstraction regarding the accuracy relative to their underlying full models (e.g., fine-grained statement-level ones) as ground truth.

In [3], a method-level dependence abstraction, called the *static-execute-after/before* (SEA/SEB), was proposed to replace traditional software dependencies (TSD) based on the system dependence graph (SDG) [2], [3]. This approach approximates dependencies among methods via control flows using the interprocedural control flow graph (ICFG) and was shown to as almost precise as static slicing based on the TSD model while coming at low costs and perfect (100%) recall. Later, the SEA was also applied to static impact analysis proven more accurate than peer techniques [12] while improving regression test selection and prioritization [13] as well.

However, previous studies of the accuracy of SEA/SEB either targeted procedural programs [3] or focused on backward dependencies based on the SEB only [6]. The remaining relevant studies addressed the accuracy of SEA-based forward dependencies, with some indeed on object-oriented programs, yet the accuracy of such dependencies was assessed either not at the method level, but at class level only [14]; or relative to ground truth not based on TSD, but on repository changes [15] or programmer opinions [12], and in the specific application context of impact analysis. While forward dependence analysis is required by many dependence-based applications, including impact analysis that SEA/SEB has been mainly applied to, the accuracy of this abstraction with respect to TSD, for object-oriented programs in particular, remains unknown.

In this paper, we present an alternative method-level dependence abstraction called the method dependence graph (MDG). For one thing, the MDG directly models dependencies among all methods of a program, with detailed dependencies within methods abstracted away, and does so in a context-insensitive manner, thus it is more efficient than TSD [2], [11]. On the other hand, this abstraction captures whole-program control and data dependencies, including optionally those due to exception-driven control flows [16], 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-effectiveness dependence abstraction over existing alternative options.

We implemented the MDG and applied it to impact analysis for Java[1], which are both evaluated on five non-trivial subject programs. We computed the accuracy of the MDG for approximating forward dependencies in general and the cost-effectiveness in supporting impact analysis; we also compared the accuracy and efficiency of the MDG relative to TSD-based forward slicing against SEA. Our results show that the MDG can approximate TSD for perfect recall (100% recall) and high precision (85-90% mostly) with great efficiency for method-level forward dependencies. Our study also demonstrated that the MDG abstraction can significantly enhance the cost-effectiveness of impact analysis over the SEA approach. While not a program representation like the MDG, SEA was originally proposed as a substitute for TSD. Thus, for brevity, we refer as SEA to both the abstraction approach based on

[1]Downloads are publicly available at http://nd.edu/~hcai/mdg

SEA relations and the SEA analysis itself when comparing them to the MDG and MDG-based analysis.

In summary, the main contributions of this paper include:

- A program dependence abstraction, called the MDG, that can approximate TSD more accurately than existing options with practical efficiency (Section III).
- An implementation of the MDG and an impact-analysis tool based on it (Section IV-A1).
- An evaluation of the MDG that assesses its accuracy relative to TSD-based forward slicing and its cost-effectiveness for impact analysis over the latest peer approach SEA (Section IV).

II. Motivation and Background

Our work was primarily motivated by improving the cost-effectiveness of forward dependence analysis that directly supports dependence-based impact analysis [5], [17] and other software-evolution tasks [18]. The need for more cost-effective impact-analysis techniques has been extensively investigated previously (e.g., [4], [7], [19]–[21]) and stressed recently [8].

A. Impact Analysis

Impact analysis is a crucial step during software evolution [5], [18], 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 a large body of research invested [17], today's impact-analysis techniques still face many challenges, most of which can be reduced to the struggle between the cost and effectiveness of the techniques or their results [4], [19], [21], [22].

In this context, we lately developed DIVER [7], [8], a dynamic impact analysis that was 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 information in that graph, the impact computation based on method execution traces generated from the test inputs. However, during its post-processing phase, intraprocedural dependencies 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 [8]). Therefore, for hybrid analysis using method-level execution data only, intraprocedural dependencies can be abstracted away.

A few approaches devoted to abstracting program dependencies to method level exist [11], [23] which, however, are as heavyweight as or even more than the TSD model [2] 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 suffered 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 it *directly* models method-level dependencies to capture only necessary information used by the dynamic analysis.

The applicability of dynamic impact analysis is constrained by the availability, and quality, of program inputs (hence those of the dynamic data), though. When such inputs are not available, impact analysis would be performed using 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]. Such analyses input a program and a query (a method for which impacts are queried), and add all methods that possibly statically execute after the query into its impact set as the output. Yet, intuitively this approach can be very imprecise because of its highly conservative nature, as discussed below.

B. The Static Execute After (SEA)

The static-execute-after relation is defined as the union of *call*, *return-into*, and *sequentially follow* relations, all considered transitively [14]. 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, against widely used object-oriented software in particular (only differences between forward and backward dependence sets based on SEA/SEB were reported in [6], 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 (of 100% recall), ignoring the analysis of them can naturally lead to false-positive dependencies. And 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.

III. The Method Dependence Graph

This section first gives a description and definition of the MDG, and then presents the algorithms for constructing it on a given input program. We use both graph and code examples for illustration.

A. Overview

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

```
public class A {
  static int g; public int d;
  String M1(int f, int z) {
    int x = f + z, y = 2, h = 1;
    if (x > y)
      M2(x, y);
    int r = new B().M3(h, g);
    String s = "M3val: " + r;
    return s;}
  void M2(int m, int n) {
    int w = m - d;
    if (w > 0)
      n = g / w;
    boolean b = C.M5(this);
    System.out.print(b);}}
```

```
public class B {
  static short t;
  int M3(int a, int b) {
    int j = 0;
    t = -4;
    if ( a < b )
      j = b - a;
    return j;}
  static double M4() {
    int x = A.g, i = 5;
    try {
      A.g = x / (i + t);
      new A().M1(i, t);
    } catch(Exception e) { }
    return x;}}
```

```
public class C {
  static boolean M5(A q) {
    long y = q.d;
    boolean b = B.t > y;
    q.d = -2;
    return b;}
  static void M0(String[] s){
    int a = 0, b = 3;
    A o = new A();
    String s = o.M1(a, b);
    double d = B.M4();
    String u = s + d;
    System.out.print(u);
  }
}
```

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

ordinary control flows, the MDG also considers those caused by exception-driven control flows [7], [16]. As such, the MDG aims to *directly* represent data and control dependencies *among methods* as attempted in [11], [23] while ignoring unnecessary statement-level details.

To further define 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 concretely, 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 the level of the port is necessary for easily deriving more precise *transitive* method-level dependencies on the MDG than otherwise if directly connecting among the nodes with single edges only.

Some of the above terms, such as (incoming/outgoing) port and dependencies, were previously used for presenting

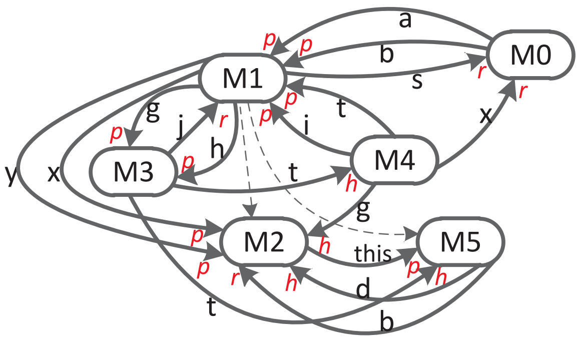

Fig. 2: The method dependence graph (MDG) of program E.

DIVER [7]. In comparison, instead of simply *renaming* vertices on interprocedural dependence edges to such terms after a detailed dependence graph was constructed through a whole-program statement-level data and control flow analysis, as we did in DIVER, here we define the MDG based on these terms *before* constructing it, and use them to guide the construction of the MDG. Thus, the MDG construction takes less time than building that fine-grained dependence graph would cost.

More specifically, we find the ports explicitly without computing interprocedural dependencies beforehand, but first locating the ports of each method, and then connecting them among methods based on type matching. In addition, we compute intraprocedural dependencies just for connecting incoming to outgoing ports *inside* methods, and *discard them afterward*—they are not included in the MDG.

To facilitate the description of the MDG and the design of its application analyses (e.g., impact analysis based on the MDG), we continue to classify interprocedural data dependencies into three categories as in [7]: *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).

2) 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*).

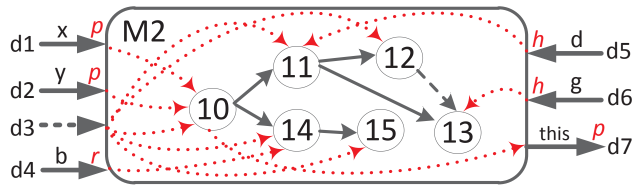

Fig. 3: Statement dependencies in `M2` used to find *summary* dependencies of outgoing on incoming dependencies in the MDG. These statement-level dependencies are *discarded* after analyzing the method.

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

```
1:  $G$ := empty graph // start with empty MDG of $P$
2:  $IP$ := $OP$ := $\emptyset$   // maps of methods to 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 $z \in OP[m]$ do
8:          add $\{\langle z, z' \rangle \mid \exists m'$ s.t. $z' \in IP[m'] \wedge data\_dep(z,z')\}$ to $G$
9:      MATCHINTERCDS($G$, $unhandled$, $m$, $IP$, $OP$)
10:     $pdg$ := GETPDG($m$)
11:     for each port $z \in IP[m]$ do
12:         add $\{\langle z, z' \rangle \mid z' \in OP[m] \wedge reaches(z,z',pdg)\}$ to node $G_m$
13: return $G$
```

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`. `M2` has also an outgoing DD edge to `M5` as it calls `M5` with parameter `this`.

Figure 3 shows the original statement-level dependencies *within* `M2` (i.e., its program dependence graph (PDG) [24]) 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 eariler, the MDG does not keep the PDGs but these summary dependencies only.

B. 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 [2], only summaries of reaching definitions, reachable uses, and control dependencies for *call sites* and *exit nodes* are kept in memory after each intraprocedural analysis. The MDG construction algorithm 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 the method-level information required by the MDG is kept.

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
- 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) according to 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 [24]. With this information and the MDG edges, a client analysis such as 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.

2) Construction Algorithm: Algorithm 1 describes the process for building an MDG. We use these helper notations: a caller (resp. call) site *crs* (resp. *cs*) is a tuple $\langle m, s\rangle$ where m is the caller (resp. 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 the 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, all possible exceptions for safety. 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 (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 [24] (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 G (line 12).

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 $\{D(a, cs) \mid a \in actual\_params(cs)\}$ to OP[m]
 4:         add $\{U(f, m') \mid f \in formal\_params(m')\}$ to IP[m']
 5: if $return\_type(m) \neq void$ then
 6:     add $\{D(rs) \mid rs \in return\_sites(m)\}$ to OP[m]
 7:     for each caller site $crs$ of $m$ do
 8:         add $\{U(crs.s, rs) \mid rs \in return\_sites(m)\}$ 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]
```

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.

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

```
 1: add the entry of $m$ to IP[m]   // the entry represents all CD
    targets for callers
 2: for each edge $\langle h, t \rangle$ 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]
```

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 [24]. 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 [25]). Then, all multi-target call sites in m are added to the *OP*s 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).

IV. EMPIRICAL EVALUATION

We evaluated our technique as a dependence abstraction in general and its application to impact analysis in particular. For that purpose, 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 SEA. Accordingly, we seek to answer the following three research questions:

RQ1 How accurately can the MDG and SEA abstract the full TSD model in terms of approximating forward dependencies?

RQ2 Can the dependence abstractions (the 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?

A. Experiment Setup

We briefly discuss key implementation issues and describe the subject programs used for obtaining the following empirical results. All our studies were performed consistently on a Linux workstation with a Quad-core Intel Core i5-2400 3.10GHz processor and 8GB DDR2 RAM.

1) Implementation: We implemented the MDG, SEA, impact analysis tools based on the two abstractions, and the method-level TSD-based forward static slicer all on top of our dependence-analysis and instrumentation system DUA-FORENSICS [26], which is built on the Soot bytecode analysis framework [27]. To compute control dependencies, including those due to exception-driven control flows, we used the *exceptional control flow graph* (ExCFG) provided by Soot as recently did in [7], [28] as well.

The ExCFG was also employed to create the interprocedural component control flow graph (ICCFG) [3], [14], on which SEA was implemented using the *on-demand* algorithm presented in [6]. For static slicing, we directly used the context-sensitive forward static slicer as part of DUA-FORENSICS with results lifted up to method level. Both the slicer and SEA implementations utilized the same call graph facilities given by Soot with the rapid type analysis applied. More details regarding the slicer, such as points-to analysis and library-call modeling, can be found in [26].

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 precisely, 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 the set of all methods that are in the SEA relation with that query.

TABLE I: 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
Jaba	bytecode analyzer	37,919	3,332

2) Subject Programs: We selected five Java programs of diverse application domains and sizes for our evaluation. Table I lists the basic characteristics of these subject programs, including the number of non-blank non-comment lines of Java code (*#LOC*) and number of methods (*#Methods*) defined in each subject. The first four subjects are all obtained from the SIR archive [29], for which we picked the first version available of each. The Jaba program was received from its authors.

B. Experimental Methodology

This main goal of our study is to addresses the accuracy of the MDG against both the SEA approach and 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 these two abstraction models and the static impact analysis techniques based on them. We also study the efficiency of all these approaches.

For this study, we applied the MDG- and SEA-based static impact analysis tools, and the method-level TSD-based forward static slicer, to each of the five 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 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 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.

Second, we computed the forward-dependence querying time costs of the MDG, the SEA, and the forward static slicing. We also 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 [30], to assess the statistical significance of mean difference in each of the above two metrics between the two abstraction approaches against the method-level static forward slicing. For the statistical test, we adopted a confidence level of 95% with the null hypothesis for no difference in the means. We also report the significance over all subjects when applicable by combing the per-subject p-values using the Fisher method [31].

C. Results and Analysis

1) Accuracy of dependence abstractions (the MDG and SEA versus TSD): Figure 4 shows the precision results of the two abstraction approaches, as listed on the x axis, in approximating forward dependencies relative to 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 for the data from one of the two approaches with that subject.

Each of the boxplots includes five components: the maximum (upper whisker), 75% quartile (top of middle box), 25% quartile (bottom of middle box), 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 very high precision 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% (the lowest, with XML-security) to 98% (the highest, with NanoXML). For NanoXML and Jaba, the precision was over 95% for 75% of queries, and for Schedule1 it was also as high as 90%. In these cases, we found many queries that share the same dependence sets, possibly due to the existence of dependence clusters as previously investigated [13]. 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 for the worst-case subject XML-security, was Ant, where 25% of queries had forward dependence sets 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—SEA did see such trend, although not constantly. Except for the same worst case as has been seen by the MDG (with XML-security), both the other two largest subjects had generally much lower precision from SEA than what the MDG gave. Besides Schedule1 and NanoXML, for which it was as almost precise as the MDG, SEA produced results of less than 5% precise for 25% of queries in other three larger subjects. For XML-security, in particular, the SEA 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 certain samples of such cases and found that in most of the resulting dependence sets only one was true positive: the query itself. While the most possible common reason was the conservativeness of both ab-

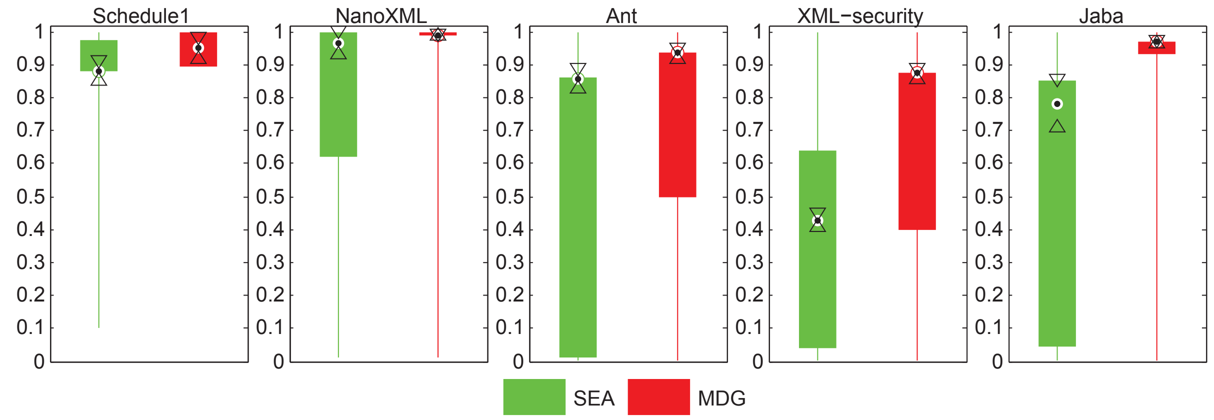

Fig. 4: Precision of the MDG and SEA relative to TSD-based forward slices as the ground truth (with constantly 100% recall).

straction approaches, the fact that the MDG has a lot less such bad cases than SEA was more probably due to their different underlying technique in nature: the MDG models both data and control dependencies, which tends to be less conservative than SEA which considers roughly control flows only.

Supplementary to the bloxplots, the left three columns of Table II gave another statistics, the means, of the precision results. 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. Comparison between the MDG and SEA reveals that both had an overall similar trend in the fluctuation of means across the five subjects: where the MDG had relatively lower mean precisions, did the SEA too. Nonetheless, the p-values (the fourth column) show statistical significant differences quite strongly in the means between the two approaches, further confirming the advantage of the MDG over SEA on top of the significance in medians shown by the comparison intervals on the boxplots of Figure 4.

Over the total data points of all subjects, the MDG had a precision of 72% on average, significantly (with p close to 0) higher than the 49% precision archived by SEA. Note that such means were largely skewed as discussed above and, thus, in most cases, users should anticipate much higher precision from the MDG than from SEA. Additionally, for the SEA, in contrast to previous accuracy studies that reported very high precision from it, at class level [14] or at method level but with procedural programs (in C) considered only [3], 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.

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

2) Efficiency of dependence abstractions (MDG and SEA versus TSD): The rest of Table II focuses on the efficiency of the two abstraction approaches versus the TSD model, including the time in seconds (*Abstraction time*) consumed by building the underlying graphs (*ICCFG* used by the SEA and the MDG), 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 abstraction models.

In terms of the graph construction time, since the MDG as a program representation contains much finer-grained information than the ICCFG, the MDG approach costs always more than the SEA, as we expected. Yet, in all cases, the costs were at most five minutes, which should be quite affordable. Note that the costs did not increase with subject sizes, implying that users would not necessarily expect increasingly higher costs for subjects of growing sizes. Intuitively, the internal logic complexity of programs is an additional factor. Moreover, this phase is required only once for all possible impact-set queries, for the single program version the client analysis works with at least. Finally, in case of larger overheads, this step could be included as part of nightly build in practice.

Also, the querying-time results to the right of the table show that the higher one-time costs of the MDG approach were easily paid off: compared to SEA, the MDG costed constantly less in terms of the mean querying time and with smaller variations. In addition, the p-values tell that, for all individual queries, the MDG was more efficient than SEA with strong significance in all cases. In all, over all subjects, the mean querying cost on the MDG is 53ms, less than half of that incurred by SEA.

Compared to the cost of the full TSD model (the last four columns), however, both abstraction models appeared to be much cheaper. For the two smallest subjects, querying the method-level forward dependencies was 100x faster on the two abstraction models than on the full TSD model; for other subjects, the speedup over TSD was almost 1000x. As an

TABLE II: PRECISION MEANS OF THE MDG VERSUS THE SEA AND TIME COSTS OF BOTH RELATIVE TO FORWARD SLICING

Subject	Mean precision			Abstraction 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)
Jaba	0.58	0.84	1.96E-33	28	302	213 (201)	121 (221)	3.58023E-10	444,188 (801,631)
Overall	**0.49**	**0.72**	**6.86E-195**	**14.8**	**104.4**	**131.4 (294.1)**	**53.3 (66.7)**	**1.95E-35**	**55,737.9 (241,084.9)**

example, the huge variations of the TSD querying costs suggest that, in some cases, a single query can take as long as a few hours, as we experienced in experimentation. Between the two abstraction approaches, on average over all queries, SEA costs 1.47% of the per-query time incurred by the TSD-based forward slicing, in contrast to 0.44% by the MDG.

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

3) Cost-effectiveness for static impact analysis (MDG versus SEA): From the foregoing comparisons, it has been seen that the MDG as a TSD approximation approach is generally much more precise than the SEA alternative, for approximating forward dependencies at least. The numbers in Table II also suggest that the MDG provides significantly faster dependence-set querying than SEA too. In addition, for the cases we studied, both approaches were confirmed to be safe, always giving perfect recall relative to the fine-grained TSD-based forward 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 implies that, besides the graph building overhead which is a one-time cost and mostly quite practical, the MDG-based static impact analysis is potentially more cost-effective than the SEA-based analysis.

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

D. Threats to Validity

The main *internal* threat to the validity of our results is the possibility of 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 infrastructures, 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 of our implementations using both the example program and two subjects Schedule1 and NanoXML for randomly sampled queries. An additional *internal* threat is the possibility of 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 obtained 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 seven years. Second, most of the core facilities that could affect the accuracy of this slicer are directly adopted or extended from the Soot framework, which is a static-analysis platform widely used by many researchers. Finally, since those core facilities are shared by our implementation of the slicer, the MDG, and 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 study subjects. This set of five Java 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. 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 [6]).

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 exactly 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 (the MDG and SEA) 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 distribution of underlying data points (e.g., normality). Additionally, we collected and analyzed data points for all methods as queries in each subject in order to avoid relevant biases.

V. RELATED WORK

We address two major categories of previous work related to ours: dependence abstraction and impact analysis.

A. Program Dependence Abstraction

Most existing dependence-abstraction approaches were designed for specific applications, and also mostly did not directly model or subsume complete method-level (data and control) dependencies. For instance, the *program summary graph* [9] was originally developed to speed up interprocedural data-flow analysis, which is similar to the data dependence abstraction part of the MDG but is built based on program call structure only and, thus, can be not only imprecise but also unsafe from the perspective of approximating a full program dependence model.

Particularly targeting impact analysis, several abstract dependence models were proposed, such as the *lattice of class and method dependence* [32] and the *influence graph* [10]. These abstractions consider only partial dependencies: The former only captures 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 latter models only data dependencies among methods in an overly conservative manner while ignoring analysis of intraprocedural dependencies, which has been shown to be highly imprecise (precision close to a transitive closure on the ICFG) [10]. The RWSets tool in [33] 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) [11], which is built by first creating the entire SDG and then simplifying statement-level edges thereof. In [23], an extended TSD model is described to generalize the definitions of interprocedural dependencies, directly at both statement and procedure levels. Directly derived from or developed atop the SDG [2], these models are at least as heavyweight as the underlying TSD model itself.

The SEA/SEB abstraction [3] 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 [19]. Developed also for interprocedural data-flow algorithms, the *program super graph* [34] 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 [35] is proposed to visualize CFGs for program comprehension, which also simplifies each intraprocedural CFG as by the SEA/SEB.

In contrast to the above approaches, 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 dismisses 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.

On the other hand, it is worth noting that this work is constrained to examining dependence abstractions with respect to the TSD model, while, as previous work revealed and studied [11], [14], hidden dependencies that cannot be captured by TSD also exist, especially in object-oriented software. And other researchers have shown that the SEA approach can help discover those dependencies [14]. Therefore, it would be also of interest to investigate in that regard using the MDG and in comparison to the SEA/SEB.

When developing the MDG, we reused some terms, such as the port and data-dependence classification, from DIVER [7]. In contrast, the MDG is more compact than the dependence graph used by DIVER. 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.

B. Impact Analysis

Static impact analysis 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 [12], [13], 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 between them), thus it tends to better support impact inspection and understanding, than the SEA-based approach.

Static program slicing has been directly used for static impact analysis, but it was shown to have challenges from overly large results and/or prohibitive costs [4]. Many other static approaches exist, which utilize various types of program information, such as code structure and version repository [12], [17]. Our static impact analysis based on the MDG utilizes method-level dependencies to offer an efficient approach with a precision comparable to fine-grained static slicing.

A large body of other impact-analysis techniques has been developed but is *descriptive* [5], [17], such as SIEVE [36], CHIANTI [37], and the impact analysis based on static slicing and symbolic execution [38]. 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. Finally, in contrast to dynamic impact analysis (e.g., [7], [8]) which relies on the availability and quality of program inputs, the MDG and MDG-based impact analysis are static, thus they do not use or depend on those inputs.

VI. 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 an impact analysis tool based on the MDG and evaluated its cost and effectiveness against the SEA-based alternative. 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. Our results also suggest that the MDG can largely improve the cost-effectiveness of impact analysis over the SEA-based analysis.

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 work shows that advantage for forward dependencies and at method level only. Thus, future study may consider addressing backward dependencies and at other levels of granularity.

Acknowledgments

This work was partially supported by ONR Award N000141410037 to the University of Notre Dame.

References

[1] A. Podgurski and L. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 965–979, 1990.

[2] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.

[3] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich, "Static execute after/before as a replacement of traditional software dependencies," in *Proceedings of IEEE International Conference on Software Maintenance*, 2008, pp. 137–146.

[4] M. Acharya and 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*, May 2011, pp. 746–765.

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

[6] J. Jász, "Static execute after algorithms as alternatives for impact analysis," *Electrical Engineering*, vol. 52, no. 3-4, pp. 163–176, 2010.

[7] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of International Conference on Automated Software Engineering*, 2014, pp. 343–348.

[8] ——, "A Framework for Cost-effective Dependence-based Dynamic Impact Analysis," in *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 231–240.

[9] D. Callahan, "The program summary graph and flow-sensitive interprocedual data flow analysis," in *Proceedings of ACM Conference on Programming Language Design and Implementation*, 1988, pp. 47–56.

[10] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating influence mechanisms into impact analysis for increased precision," in *Proceedings of IEEE International Conference on Software Maintenance*, 2006, pp. 55–65.

[11] Z. Yu and V. Rajlich, "Hidden dependencies in program comprehension and change propagation," in *Proceedings of International Workshop on Program Comprehension*, 2001, pp. 293–299.

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

[13] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Impact analysis in the presence of dependence clusters using static execute after in webkit," *Journal of Software: Evolution and Process*, vol. 26, no. 6, pp. 569–588, 2014.

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

[15] J. Jász, L. Schrettner, Á. Beszédes, C. Osztrogonác, and T. Gyimóthy, "Impact analysis using static execute after in webkit," in *Proceedings of European Conference on Software Maintainance and Reengineering*, 2012, pp. 95–104.

[16] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.

[17] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, pp. 613–646, 2013.

[18] V. Rajlich, "Software evolution and maintenance," in *Proceedings of the Conference on the Future of Software Engineering*, 2014, pp. 133–144.

[19] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, May 2005, pp. 432–441.

[20] H. Cai, S. Jiang, R. Santelices, Y. jie Zhang, and Y. Zhang, "SENSA: Sensitivity analysis for quantitative change-impact prediction," in *Proceedings of IEEE Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 165–174.

[21] H. Cai and R. Santelices, "A Comprehensive Study of the Predictive Accuracy of Dynamic Change-Impact Analysis," *Journal of Systems and Software*, vol. 103, pp. 248–265, 2015.

[22] H. Cai, R. Santelices, and T. Xu, "Estimating the accuracy of dynamic change-impact analysis using sensitivity analysis," in *Proceedings of International Conference on Software Security and Reliability*, 2014, pp. 48–57.

[23] J. P. Loyall and S. A. Mathisen, "Using dependence analysis to support the software maintenance process," in *Proceedings of IEEE International Conference on Software Maintenance*, 1993, pp. 282–291.

[24] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[25] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, pp. 209–254, 2001.

[26] R. Santelices, Y. Zhang, H. Cai, and 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*, Jun. 2013, pp. 13–18.

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

[28] H. Cai and R. Santelices, "TracerJD: Generic Trace-based Dynamic Dependence Analysis with Fine-grained Logging," in *Proceedings of IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2015, 489–493.

[29] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[30] R. E. Walpole, R. H. Myers, S. L. Myers, and K. E. Ye, *Probability and Statistics for Engineers and Scientists*. Prentice Hall, Jan. 2011.

[31] F. Mosteller and R. A. Fisher, "Questions and answers," *The American Statistician*, vol. 2, no. 5, pp. 30–31, 1948.

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

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

[34] E. M. Myers, "A precise inter-procedural data flow algorithm," in *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1981, pp. 219–230.

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

[36] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2006, pp. 241–252.

[37] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, Oct. 2004, pp. 432–448.

[38] N. Rungta, S. Person, and J. Branchaud, "A change impact analysis to characterize evolving program behaviors," in *Proceedings of IEEE International Conference on Software Maintenance*, 2012, pp. 109–118.