Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA

{term|orso|harrold}@cc.gatech.edu

ABSTRACT

As software evolves, impact analysis estimates the potential effects of changes, before or after they are made, by identifying which parts of the software may be affected by such changes. Traditional impact-analysis techniques are based on static analysis and, due to their conservative assumptions, tend to identify most of the software as affected by the changes. More recently, researchers have begun to investigate dynamic impact-analysis techniques, which rely on dynamic, rather than static, information about software behavior. Existing dynamic impact-analysis techniques are either very expensive—in terms of execution overhead or amount of dynamic information collected—or imprecise. In this paper, we present a new technique for dynamic impact analysis that is almost as efficient as the most efficient existing technique and is as precise as the most precise existing technique. The technique is based on a novel algorithm that collects (and analyzes) only the essential dynamic information required for the analysis. We discuss our technique, prove its correctness, and present a set of empirical studies in which we compare our new technique with two existing techniques, in terms of performance and precision.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools, Tracing*;

General Terms: Algorithms, Experimentation

Keywords: Impact analysis, dynamic analysis, software maintenance

1. INTRODUCTION

As software evolves, changes to the software can have unintended or even disastrous effects [8]. Software change impact analysis estimates the potential effects of changes before or after they are made. Applied before modifications, impact analysis can help maintainers estimate the costs of proposed changes and select among alternatives.

Applied after modifications, impact analysis can alert engineers to potentially affected program components that require retesting, thus reducing the risks associated with releasing changed software.

1.1 Dynamic versus Static Impact Analysis

Traditional impact-analysis techniques (e.g., [2, 9, 14, 15, 16]) rely on static analysis to identify the *impact set*—the subset of elements in the program that may be affected by the changes made to the program.

Although static-analysis-based techniques can safely estimate the impact of changes, their conservative assumptions often result in impact sets that include most of the software. For example, in previous work, we found that impact analysis based on static slicing often identified, for the software considered, impact sets that included more than 90% of the program [10]. Such impact sets make the results of impact analysis almost useless for other software-engineering tasks. For example, regression-testing techniques that use impact analysis to identify which parts of the program to retest after a change would have to retest most of the program.

The problem with sound static-analysis-based approaches is twofold. First, they consider all possible behaviors of the software, whereas, in practice, only a subset of such behaviors may be exercised by the users. Second, and more importantly, they also consider some impossible behaviors, due to the imprecision of the analysis. Therefore, recently, researchers have investigated and defined impact-analysis techniques that rely on dynamic, rather than static, information about program behavior [3, 6, 7, 10]. The dynamic information consists of execution data for a specific set of program executions, such as executions in the field, executions based on an operational profile, or executions of test suites.

We define the *dynamic impact set* to be the subset of program entities that are affected by the changes during at least one of the considered program executions. *Dynamic impact analysis* is the analysis that computes (approximates) dynamic impact sets. In the rest of the paper, we use the terms dynamic impact set and impact set interchangeably, unless otherwise stated.

1.2 Existing Approaches

To the best of our knowledge, two main dynamic impact analysis techniques have been defined in the literature: `CoverageImpact` [10] and `PathImpact` [6, 7]. (Breech and colleagues' technique [3] is a variant of `PathImpact` that
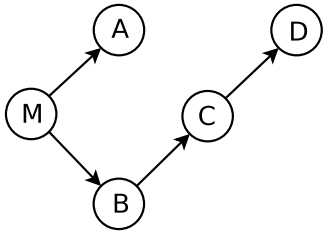
Figure 1: Call graph of an example program P

$$\boxed{\text{M}_e\ \text{A}_e\ \text{A}_r\ \text{A}_e\ \text{A}_r\ \text{B}_e\ \text{C}_e\ \text{C}_r\ \text{B}_r\ \text{B}_e}$$
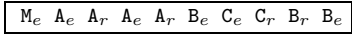
Figure 2: Trace t for program P

computes the impact sets for all methods online.) We illustrate the differences between these two techniques with an example. Consider program P, whose call graph[1] is shown in Figure 1, and an execution of P, whose trace t is shown in Figure 2. The trace consists of a list of method entries and returns. For a given method X, an occurrence of X_e in the trace indicates a call to X at that point of the execution. Analogously, an occurrence of X_r in the trace indicates a return from X at that point of the execution. In the following discussion, we refer to X_e as a *method-entry event* and X_r as a *method-return event*. Note that we consider return events in the most general way possible: a return event for method X occurs any time X returns into its caller regardless of how the return occurs (e.g., because of an actual `return` statement or because of an uncaught exception). The trace in Figure 2 represents an execution in which M is called, M calls A, A returns, M calls A again, A returns, M calls B, B calls C, C returns, B returns, M calls B again, and B exits, terminating the program.

`PathImpact` works at the method level and uses compressed execution traces to compute impact sets. Because the compression reduces the space cost of the approach, but does not affect the nature of the algorithm, we describe the equivalent algorithm that works on uncompressed traces.

Given a set of changes, `PathImpact` performs forward and backward walks of a trace to determine the impact set for the changes. The forward walk determines all methods called after the changed method(s), whereas the backward walk identifies methods into which the execution can return. More precisely, for each changed method X and each occurrence of X_e:

- in the forward walk, `PathImpact` starts from the immediate successor of X_e, includes every method called after X in the impact set (i.e., every method Y such that the trace contains an entry Y_e after the occurrence of X_e), and counts the number of unmatched returns;[2]

- in the backward walk, `PathImpact` starts from the immediate predecessor of X_e and includes as many unmatched methods[3] as the number of unmatched returns counted in the forward walk; and

---

[1]A *call graph* is a directed graph in which nodes represent functions or methods, and an edge between nodes A and B indicates that A may call B.

[2]Unmatched returns are returns that do not have a corresponding method entry in the part of the trace examined.

[3]Unmatched methods are method entries that do not have a corresponding return in the part of the trace examined.

- finally, X itself is added to the impact set, if it is not already there.

To illustrate, consider how `PathImpact` computes the impact set for program P (Figure 1), trace t (Figure 2), and change set $C = \{C\}$ (i.e., only method C is modified). In the forward walk, `PathImpact` starts at C_r, the successor of C_e, counts two unmatched returns (C_r and B_r), and adds B to the impact set (because of the occurrence of B_e). In the backward walk, `PathImpact` starts at node B_e, the predecessor of C_e, and, because it found two unmatched returns during the forward walk, adds the first two methods it encounters that have unmatched returns (B and M) to the impact set. The resulting impact set is $\{M, B, C\}$.

`CoverageImpact` also works at the method level, but uses coverage, rather than trace, information to compute impact sets. The coverage information for each execution is stored in a bit vector that contains one bit per method in the program. If a method is executed in the execution considered, the corresponding bit is set; otherwise, it remains unset. In our example, the execution of P would produce the bit vector shown in Table 1.

M	A	B	C	D
1	1	1	1	0

Table 1: Coverage bit vector for the execution of P that produces trace t.

`CoverageImpact` computes the impact sets in two steps. First, using the coverage information, it identifies the executions that traverse at least one method in the change set C and marks the methods covered by such executions. Second, it computes a static forward slice from each change in C considering only marked methods. The impact set is the set of methods in the computed slices.

To illustrate, consider how `CoverageImpact` computes the impact set for our example and change set $C = \{C\}$ (the same used to illustrate `PathImpact`). Because C is traversed by this execution, the methods covered by the execution (M, A, B, and C) are marked. Assuming that a traditional static slice from method C would include all methods in the program, the resulting impact set—the slice computed considering only marked methods—would be $\{M, A, B, C\}$. Note that the inclusion of method A is an imprecision: because `CoverageImpact` uses simple coverage instead of traces, it misses information about the ordering of method calls.

1.3 Our Approach

In previous work [11], we compared the precision and performance of `CoverageImpact` and `PathImpact`, both analytically and empirically. Analytically, `PathImpact` is more precise than `CoverageImpact`, as shown in our example, because it uses traces instead of coverage. However, `PathImpact` incurs much higher overhead in both time and space. Where time is concerned, `CoverageImpact` requires a constant time to update the bit vector at each method entry, whereas `PathImpact` requires a time that depends on the size of the trace analyzed so far to compress traces at both method entry and return. Where space is concerned, `CoverageImpact` requires one bit per method (i.e., its space complexity is linear in the size of the program), whereas `PathImpact`'s space cost is proportional to the size of the traces (which can be

very large). Our empirical studies confirmed that this trade-off also occurs in practice. For some programs and versions, the impact sets computed by `CoverageImpact` are significantly less precise than those computed by `PathImpact`, but `CoverageImpact` imposes orders of magnitude less overhead in both time and space across all programs and versions [11]. In fact, our studies show that the overhead imposed by `PathImpact` makes it impractical to use with all but short-running programs.

Based on these results, our goal was to define an approach that is efficient and practical (i.e., one that can be used for large, long-running programs), yet is more precise than `CoverageImpact`. Initially, we investigated a hybrid approach that (1) uses static analysis to identify groups of methods that could cause imprecision, and (2) collects additional dynamic information, such as partial traces, for those methods. However, in the process, we made some fundamental observations about the essential information that is required to perform dynamic impact analysis. These observations led us to a novel algorithm for dynamic impact analysis. This algorithm is as precise as `PathImpact` but only slightly more expensive than `CoverageImpact`.

In this paper, we present our new technique for dynamic impact analysis and demonstrate the correctness of its underlying algorithm. We also discuss our implementation of the technique for the Java language. Finally, we present a set of empirical studies in which we compare our new technique with techniques `PathImpact` and `CoverageImpact`. The studies confirm that our technique is practical: it is almost as efficient as `CoverageImpact` and as precise as `Path-Impact`.

The main contributions of this paper are:

- the identification of the essential information needed to perform dynamic impact analysis;

- the definition of a generic technique, and an underlying algorithm, for efficiently collecting and analyzing this information;

- the instantiation of the technique for the Java language; and

- a set of empirical studies, performed on real programs, that show the efficiency and effectiveness of our new technique and compare it to existing approaches.

2. DYNAMIC IMPACT ANALYSIS ALGO-RITHM

In this section, we first discuss our findings on what information is essential for computing dynamic impact sets. Then, we introduce our algorithm for collecting this information efficiently during program executions and present a proof of correctness for the algorithm.

2.1 The Execute-After Relation

As we stated in Section 1.2, dynamic impact analysis computes, for one or more program changes, the corresponding dynamic impact set: the set of program entities that *may* be affected by the change(s) for a specific set of program executions. Intuitively, all entities that are executed after a changed entity are potentially affected by that change. A safe way to identify the dynamic impact set for a changed

entity e is, thus, to include all program entities that are executed after e in the considered program executions.

Therefore, to compute dynamic impact sets for a program P and a set of executions E, the only information required is whether, for each pair of entities $e1$ and $e2$ in P, $e2$ was executed after $e1$ in any of the executions in E. This binary relation, that we call *Execute After* (*EA* hereafter) can be defined for entities at different levels of granularity. For ease of comparison with existing dynamic impact analysis techniques, we formally define the EA relation for the case in which the entities considered are methods and executions are single-threaded. We will discuss the generalization of the EA relation with regard to multi-threaded executions in Section 2.3.

DEFINITION 1. *Given a program P, a set of executions E, and two methods X and Y in P, $(X, Y) \in EA$ for E if and only if, in at least one execution in E,*

1. *Y calls X (directly or transitively),*
2. *Y returns into X (directly or transitively), or*
3. *Y returns into a method Z (directly or transitively), and method Z later calls X (directly or transitively).*

The problem with existing dynamic impact analysis techniques [3, 6, 7, 10] is that they do not explicitly compute the EA relation. Instead, they infer the relation from information that is either too expensive to collect or too imprecise to provide accurate results. For example, technique `PathImpact` uses complete program traces to identify which methods are executed after a change (see Section 1.2). For another example, technique `CoverageImpact` uses coverage information combined with static slicing to approximate the information contained in complete program traces. Our investigation shows that trace information is excessive for both deriving the EA relation and performing dynamic impact analysis because it contains much unnecessary information. In the following discussion, we demonstrate that execution traces contain mostly redundant information, and we present the considerably smaller amount of information that our technique collects at runtime.

Our first finding, when analyzing the information contained in program traces such as the one in Figure 2, is that using only the information provided by method-return events unnecessarily complicates the analysis of the traces. Method-return events can be used to identify the methods into which the execution returns, but provide this information only indirectly—some form of the stack-based walk of the traces is typically required to identify such methods. To simplify the dynamic impact analysis, we collect, instead of method-return events, what we call method-returned-into events. A *method-returned-into event* for a method X, denoted as X_i, is generated when an execution returns (from any method) into X. For now, we simply assume that method-returned-into events can be easily collected. In Section 3.1, we discuss how we efficiently collect such events for Java programs. By considering only method-entry and method-returned-into events, we rewrite the trace in Figure 2 as follows:

$$\boxed{\text{M}_e \ \ \text{A}_e \ \ \text{M}_i \ \ \text{A}_e \ \ \text{M}_i \ \ \text{B}_e \ \ \text{C}_e \ \ \text{B}_i \ \ \text{M}_i \ \ \text{B}_e}$$

In the rest of the section, we use this example trace to illustrate how to capture the EA relation between any pair of methods in an execution.

Obviously, we can derive the EA relation from this complete trace. Because of the way in which we defined method-entry and method-returned-into events, we can observe the following:

> Method X executes after method Y if, in the trace, there is a method-entry or method-returned-into event for X that follows a method-entry or method-returned-into event for Y.

However, if our only goal is to derive the EA relation, the complete trace contains much unnecessary information. In fact, the above observation can be restated as follows:

> $(X, Y) \in EA$ iff at least one event for X occurs after at least one event for Y.

To assess whether at least one event for X occurs after at least one event for Y, we do not need a complete trace—it is enough to consider the first method event for Y (we refer to this event as Y_f), the last method event for X (we refer to this event as X_l), and the ordering of the two events in the trace. If an event Y_*[4] for method Y occurs before an event X_* for method X, then necessarily Y_f occurs before X_l: by definition, $Y_f \leq Y_* < X_* \leq X_l$. Conversely, if Y_f occurs after X_l, then there cannot be any X_* and Y_* such that Y_* occurs before X_*: $Y_* < X_*$ contradicts $X_* \leq X_l < Y_f \leq Y_*$.

We can thus conclude that, in general, the essential information for deriving the EA relation for an execution is, for each method, the first and the last events that the method generates in the execution. The first event for a method X always corresponds to the first method-entry event for X. The last event for a method X corresponds to the last method-entry event for X or the last method-returned-into event for X, whichever comes last. Intuitively, the first and last events for a method represent the first and the last executions of the method, where *execution* of a method means the execution of one or more of the statements in the method's body.

By considering only the first and the last events for each method, we reduce our example trace to the following sequence:

$$\boxed{M_e \quad A_e \quad A_e \quad B_e \quad C_e \quad M_i \quad B_e}$$

To simplify the discussion, in the rest of this section we use the notation for method events introduced above: for a method X, X_f indicates the first method event for X, and X_l indicates the last method event for X. Using this notation, we rewrite the above trace as follows:

$$\boxed{M_f \quad A_f \quad A_l \quad B_f \quad C_f \quad C_l \quad M_l \quad B_l}$$

Note that, because there is only one event for method C, the event appears as both first and last. This sequence contains at most two entries for each method in the program. Because this sequence lets us derive the EA relation, we refer to it as *EA sequence*. As we discussed above, the EA sequence contains the essential information needed to perform dynamic impact analysis.

---

[4]The notation Y_* and X_* indicates any event for method Y and X, respectively.

Using EA sequences, we can thus perform dynamic impact analysis as precise as an analysis performed on complete traces, while achieving significant space savings. These savings are obvious when we consider collecting dynamic information for real executions, in which methods can be executed thousands (or millions) of times. However, achieving space savings by collecting EA sequences would not be useful if, to collect them, we still need to gather complete execution traces first. Therefore, we developed an algorithm, presented in the next section, for collecting EA sequences on the fly at a cost comparable to the cost of collecting simple method coverage information.

2.2 Algorithms

One straightforward way to collect EA sequences is to use a list of events and update it (1) at each method entry and (2) every time the flow of control returns into a method after a call. The update must operate so that only the first and the last events for each method are kept in the list. Therefore, every time an event for a method X is generated, we check whether the list already contains entries for X. If not, we add both an X_f entry and an X_l entry at the end of the list. Otherwise, if there is already a pair of entries, we remove the existing X_l entry and add a new X_l entry at the end of the list. (Intuitively, we only record the first method event and keep updating the last method event.) This algorithm is space efficient—the space required never exceeds $2n$, where n is the number of methods in the program. However, it is not time efficient because for every method event generated, the event list must be searched and updated. We could eliminate the searching time by keeping a pointer to the last event for each method and by suitably updating such pointers every time a method event is generated. However, this approach needs to update up to five pointers at each event and is, thus, penalized by the memory-management overhead.

To minimize the overhead imposed by the analysis, we developed an algorithm for collecting EA sequences at runtime that is more efficient (by a constant factor) than the list approach, in terms of both time and space, and that does not incur memory-management overhead. Our algorithm is based on the use of two arrays of event timestamps, F and L. Arrays F and L are used to store the timestamp of the first and last event, respectively, generated by each method. To denote the element of array F (resp., L) for a method X, we use the notation F[X] (resp., L[X]). The timestamp is a global counter that is incremented by one at each event. Figure 3 shows the algorithm, `CollectEA`.

`CollectEA` is an on-line algorithm, whose different parts are triggered by the events that occur during a program execution. When the program starts, all elements of arrays F and L are initialized to $\perp$, and the counter is initialized to 1. $\perp$ denotes a non-numeric special value used to identify methods that have not yet been executed. (If a method has value $\perp$ at the end of the execution, then that method was not executed at all in that execution.) Every time a method M is entered, the algorithm checks the value of F[M]. If F[M] is $\perp$ (i.e., M has not yet been executed), then the algorithm sets F[M] to the current value of the counter (lines 6–8). Because, as discussed in the previous section, the last event generated by M may be a method-entry event, L[M] is also set to the current value of the counter (line 9). Finally, the counter is incremented by one (line 10).

Algorithm CollectEA
Declare: F array of first method events
        L array of last method events
        C counter
        n number of methods in the program
Output: F, L

Begin:
1: On program start **do**
2:   initialize F[i] to $\perp$, for $0 \le i < n$
3:   initialize L[i] to $\perp$, for $0 \le i < n$
4:   initialize C to 1
end

Begin:
5: On entry of method M **do**
6:   **if** (F[M] = $\perp$) **then**
7:     F[M] = C
8:   **endif**
9:   L[M] = C
10:   increment C by 1
end

Begin:
11: On control returning into method M **do**
12:   L[M] = C
13:   increment C by 1
end

Begin:
14: On program termination **do**
15:   output F, L
end

Figure 3: Algorithm CollectEA

Every time the control flow returns into method M, L[M] is updated to the current value of the counter (line 12), and the counter is incremented by one (line 13). In this way, each element in L contains the timestamp of the last time the corresponding method was (partially) executed.

To illustrate the algorithm, we show how it works on the execution producing the example trace used in Section 1.2 (see Figure 2). Table 2 shows the values of F, L, and C after each method (and program) event. The leftmost column (event) shows the program-start, method-entry, and method-returned-into events. Columns labeled F and L show, for each method, the values of the corresponding elements in the F and L arrays, respectively. Finally, the rightmost column (C) shows the value of the counter.

On program start, F, L, and C are initialized. When M is called, F[M] and L[M] are set to 1, the current value of C, and C is incremented to 2. Likewise, when A is called, F[A] and L[A] are set to 2, and C is incremented to 3. Then, the control flow returns into M, which generates an M_i event, and L[M] and C are updated accordingly. When method A is called again, F[A] is not updated (because its value is not $\perp$), L[A] is updated, and the counter is incremented. Additional updates of F, L, and C occur in an analogous way until the program terminates.

To illustrate that, for a given execution, the information in a pair of F and L arrays is equivalent to the information in an EA sequence, we demonstrate the steps to derive one from the other and vice-versa. (Note that maintainers need

event	F					L					C
	M	A	B	C	D	M	A	B	C	D	
start	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	1
M_e	1	$\perp$	$\perp$	$\perp$	$\perp$	1	$\perp$	$\perp$	$\perp$	$\perp$	2
A_e	1	2	$\perp$	$\perp$	$\perp$	1	2	$\perp$	$\perp$	$\perp$	3
M_i	1	2	$\perp$	$\perp$	$\perp$	3	2	$\perp$	$\perp$	$\perp$	4
A_e	1	2	$\perp$	$\perp$	$\perp$	3	4	$\perp$	$\perp$	$\perp$	5
M_i	1	2	$\perp$	$\perp$	$\perp$	5	4	$\perp$	$\perp$	$\perp$	6
B_e	1	2	6	$\perp$	$\perp$	5	4	6	$\perp$	$\perp$	7
C_e	1	2	6	7	$\perp$	5	4	6	7	$\perp$	8
B_i	1	2	6	7	$\perp$	5	4	8	7	$\perp$	9
M_i	1	2	6	7	$\perp$	9	4	8	7	$\perp$	10
B_e	1	2	6	7	$\perp$	9	4	10	7	$\perp$	11

Table 2: Values of F, L, and C during the example execution.

not perform these steps to obtain impact sets from a pair of F and L arrays.) To convert a pair of F and L arrays to an EA sequence, we just need to (1) order the elements of F and L (considered together and without including elements with value $\perp$) based on their value, and (2) for each method X, replace F[X] with X_f and L[X] with X_l. To convert an EA sequence to a pair of F and L arrays, we do the opposite: (1) for each method X, we replace X_f with F[X] and X_l with L[X], and (2) we assign increasing values, starting from 1, to the elements in arrays F and L, based on their position.

For example, the pair of F and L arrays for our example (shown as the last row of Table 2) would first be ordered,

$$\boxed{\text{F[M] F[A] L[A] F[B] F[C] L[C] L[M] L[B]}}$$

and then be converted as follows:

$$\boxed{M_f \ A_f \ A_l \ B_f \ C_f \ C_l \ M_l \ B_l}$$

Because arrays F and L provide the same information as an EA sequence, we can derive the EA relation from such arrays, as stated in the following lemma:

LEMMA 1. $(X, Y) \in EA \iff F[Y] < L[X]$

PROOF. To prove Lemma 1, we leverage three characteristics of our algorithm:

1. Counter C increases monotonically each time a method event occurs.

2. For each method X, F[X] is set only once, to the then-current value of counter C, at the first method-entry event for X.

3. For each method X, L[X] is set to the then-current value of counter C every time a method event for X occurs.

Our proof proceeds in two parts, by first showing that
    $(X, Y) \in EA \Rightarrow F[Y] < L[X]$ (1)
and then showing that

    $F[Y] < L[X] \Rightarrow (X, Y) \in EA$ (2)

We first prove part (1). According to the definition of EA relation (Definition 1), there are three cases in which $(X, Y) \in EA$

In the first case (Y calls X), a Y_e event is generated at timestamp t_1 and an X_e event is generated at timestamp $t_2 >$

t_1. At the end of the execution, because of Characteristics 1, 2, and 3, F[Y] is either t_1 (if Y_e is the first entry event for Y) or a value less than t_1 (otherwise), and L[X] is either t_2 (if X_e is the last method event for X) or a value greater than t_2 (otherwise). Thus, we conclude that F[Y] < L[X] in this case.

In the second case (Y returns into X), a Y_e event is generated at timestamp t_1 (when X calls Y directly or transitively) and an X_i event is generated at timestamp $t_2 > t_1$ (when Y returns). As for the previous case, F[Y] $\leq t_1$, and L[X] $\geq t_2$. Thus, F[Y] < L[X] also in this case.

In the third case (Y returns into a method Z that then calls X), a Y_e event is generated at timestamp t_1 (when Z calls Y) and an X_e event is generated at timestamp $t_2 > t_1$ (when Z calls X). As for the previous cases, F[Y] $\leq t_1$, and L[X] $\geq t_2$, and thus F[Y] < L[X] also in this case.

Because (X, Y) $\in$ EA implies F[Y] < L[X] in all three cases, part (1) of the Lemma holds.

Next, we prove part (2). This part follows directly from the meaning of arrays F and L: if F[Y] < L[X], then the first (partial) execution of method Y precedes the last (partial) execution of method X—X executes after Y. $\square$

Now that we have shown that the `CollectEA` algorithm correctly captures the EA relation among methods for a given execution, we discuss how our technique uses such information to compute dynamic impact sets. To compute the dynamic impact set for a changed method, our technique includes every method whose timestamp in L is greater than or equal to the timestamp in F for the changed method. In the case of more than one changed method, our technique just needs to compute the impact set for the changed method with the least timestamp in the F array (*CLT* hereafter). By definition, the impact set for CLT is a superset of the impact set computed for any of the other changed methods: any other changed method X has a greater timestamp than CLT and, thus, the set of methods executed after X is a subset of the set of methods executed after CLT. More formally, given a set of changed methods CHANGED, our technique identifies CLT and computes the dynamic impact set for CHANGED as follows:

$$\text{CLT} = X \mid F[X] \leq F[Y],\ X,Y \in \text{CHANGED}$$

$$\text{impact set for CHANGED} = \{\ X \mid L[X] \geq F[CLT]\ \}$$

To illustrate this, consider our example execution and a CHANGED set that consists of A and C. In this case, CLT is method A, and the dynamic impact set for CHANGED is {M, A, B, C}. Note that changed methods that were not executed (i.e., methods whose timestamps are $\bot$ at the end of the execution) are not considered. In the case of multiple executions (i.e., multiple EA sequences), the impact set is computed by taking a union of the impact sets for the individual executions.

LEMMA 2. *The dynamic impact sets computed as described include (1) the modified methods and (2) all and only methods that are (partially) executed after any of the modified methods.*

PROOF. By definition, our technique computes dynamic impact sets with the following property:

impact set = { X | L[X] $\geq$ F[Y] for any modified method Y}

Lemma 2 follows immediately from Lemma 1 and from the above property. $\square$

The space complexity of `CollectEA` is $O(n)$, where n is the number of methods in the program, because the algorithm needs two arrays, each of size n, and a counter. Compared to approaches that use traces, our algorithm achieves dramatic savings in terms of space because program traces, even if compressed, can be very large. For example, in our previous work, we collected traces on the order of 2 gigabytes, even for relatively small programs [11]. The time overhead of `CollectEA` is a small constant per method call. At each method entry, the algorithm performs one check, one increment, and at most two array updates. Every time the control returns into a method, the algorithm performs one array update and one increment.

2.3 Multi-Threaded Executions

In multi-threaded executions, one method can be executed not only before or after another method, but also concurrently. According to the definition of dynamic impact analysis, any method (or part thereof) that is executed after a changed method is potentially affected by the change. Therefore, any method that is executed concurrently with a changed method is also potentially affected by the change because of possible interleaving of threads. Unfortunately, method-entry, and method-returned-into events are not enough to identify affected methods in these cases.

To illustrate, consider a multi-threaded program in which method A is entered at time t_1 and exited at time t_2, method B is entered at time t_3 and exited at time t_4, and $t_1 < t_3 < t_2$. In such a case, A and B are executed in parallel, and a possible sequence of events is (assuming that methods A and B are invoked by two methods X and Y, respectively):

$$\boxed{\ \texttt{...}\quad \texttt{A}_f\ \ \texttt{A}_l\ \ \texttt{B}_f\ \ \texttt{B}_l\ \ \texttt{X}_l\ \ \texttt{Y}_l\ }$$

If method B is a changed method, the above sequence does not give us enough information to identify A as possibly affected by B because it only appears before B. To address this problem, and account for multi-threaded executions, we modify our algorithm `CollectEA` as follows. We still use one pair of arrays F and L with a global counter, but we also collect method-return events. Method-return events let us identify whether one method in a thread is exited before or after the entry of another method in another thread. The algorithm treats method-return events in the same way in which it treats method-return-into events. For the example above, the trace would therefore change as follows:

$$\boxed{\ \texttt{...}\quad \texttt{A}_f\ \ \texttt{B}_f\ \ \texttt{A}_l\ \ \texttt{X}_l\ \ \texttt{B}_l\ \ \texttt{Y}_l\ }$$

We then compute the impact sets from arrays F and L in the same way as previously described. The impact sets obtained in this way are safe, in the sense that no method that is not in the impact set for a change can be affected by that change.

3. EMPIRICAL STUDIES

To evaluate our technique, we developed a prototype tool, called EAT (Execute-After Tool), and conducted a set of empirical studies. In the studies, we investigate the following research questions:

RQ1: How much overhead does the instrumentation required by our technique impose, in practice, on the programs

under analysis compared to the more efficient of the two existing techniques?

RQ2: How much does our technique gain, in terms of precision, with respect to a technique that does not collect trace information?

3.1 The Tool: EAT

EAT is a tool written in Java that consists of three main components: (1) an instrumentation module, (2) a set of runtime monitors, and (3) an analysis module.

3.1.1 Instrumentation Module

The instrumentation module uses INSECT [5] to instrument the program under analysis by adding probes that produce method events. In Section 2, we assumed that method events can be easily produced. The way these events are produced in practice depends on the programming language that is targeted by the analysis. Because the subjects of our studies are Java programs, we discuss how to collect the events for the Java language.

Collecting method-entry events is straightforward. We simply instrument each method immediately before the first statement with a probe that generates an event with an attribute. The attribute is the numeric identifier for the method in which the event is generated.

Collecting method-returned-into events is more complicated because, in Java, there are three ways in which a method X can return into another method Y:

1. Normal return: X returns into Y because of a `return` statement or simply because X terminates. In this case, the execution continues at the instruction in Y that immediately follows the call to X.

2. Exceptional return into a catch block: while X executes, an exception is thrown that is not caught in X but is caught in Y. In this case, the execution continues at the first instruction in the catch block in Y that caught the exception.

3. Exceptional return into a finally block: while X executes, an exception is thrown that is not caught in X and not caught in Y, but Y has a finally block associated with the code segment that contains the (possibly indirect) call to X. In this case, the execution continues at the first instruction in the finally block in Y.

The instrumentation for a Java program for collecting method-returned-into events must handle these three cases. To this end, the instrumentation module instruments each call site by adding (1) a probe immediately before the instruction that follows a method call, (2) a probe before the first instruction of each catch block (if any) associated with the code segment that contains the call, and (3) a probe before the first instruction of the finally block (if any) associated with the code segment that contains the call. Each of these probes generates an event and attaches to the event, in the form of an attribute, the numeric identifier for the method in which the event is generated.

Note that the program instrumented in this way may generate some redundant method-returned-into events, but the correctness of the algorithm is preserved. For example, if method Y returns normally into method X, but there is a finally block in X associated with the code segment that contains the call to Y, then two probes will be triggered, which

generate two X_i events: one after the call and one in the finally block (which would be executed anyway). Every time an X_i event is duplicated, the first event produced is simply discarded when the second event occurs (i.e., the value of element L[X] is set to the new value of the counter), which is correct because the goal is to record the last time the method is executed. Such events are produced only in a few cases and, moreover, only require the update of one array element and the increment of a counter (duplication only occurs for method-returned-into events). Therefore, their impact on the efficiency of the approach is unnoticeable. Obviously, a more sophisticated instrumentation could avoid the production of these duplicated events, but the additional overhead would hinder the practicality of the approach.

The other two events required by our approach, program start and program termination, are already provided by INSECT [5], so we simply enable them when instrumenting.

3.1.2 Monitors

The monitors are static methods that implement the four parts of the algorithm shown in Figure 3. The monitors initialize, update, and output the F and L arrays during program executions.

Leveraging INSECT functionality, we link the events generated by the probes with the appropriate monitors. Therefore, when a method event is generated, INSECT calls the appropriate static method and passes the event attribute (i.e., the identifier of the method in which the event was generated) as a parameter. When a program event is generated, which happens only at program start and program termination, INSECT simply calls the appropriate method with no parameter.

3.1.3 Analysis Module

The analysis module inputs the arrays produced by the monitors and the change information and outputs dynamic impact sets. To compute the impact sets, the analysis module uses the approach described in Section 2.2.

3.2 Experimental Setup

3.2.1 Subject programs

As subjects for our study we utilized several releases of two programs—JABA and SIENA—summarized in Table 3. The table shows, for each subject, the number of versions (*Versions*), the number of classes (*Classes*), the number of methods (*Methods*), the number of non-comment lines of code (*LOC*), and the number of test cases in the subject's test suite (*Test Cases*). The number of classes, methods, and lines of code is averaged across versions.

Program	Versions	Classes	Methods	LOC	Test Cases
SIENA	8	24	219	3674	564
JABA	11	355	2695	33183	215

Table 3: Subject programs

SIENA [4] is an Internet-scale event notification middleware for distributed event-based applications. JABA[5] is a framework for analyzing Java programs. For both subjects, we extracted from their CVS repositories consecutive versions from one to a few days apart.

---

[5]`http://www.cc.gatech.edu/aristotle/Tools/jaba.html`

3.2.2 Method and Measures

Because dynamic impact analysis requires dynamic information, we used the test suites for the subjects as input sets. The test suite for SIENA was created for earlier experiments [12]. The test suite for JABA, which was also available through CVS, was created and used internally by the program developers. This test suite is divided into two parts: short tests and long tests. In our previous work, we used only the set of short tests (125 test cases) [11]. In the studies presented in this paper, we also use the set of long tests (90 test cases). For ease of comparison with previous work, we label the results for the short and the long tests differently, as *Jaba* and *Jaba-long*, respectively.

As change sets for use in assessing impacts, we used the actual sets of changes from each version of our subject programs to each subsequent version. To compute such changes, we used JDIFF, a differencing tool for Java programs developed by the authors [1].

In the description of the studies, we refer to our technique as `CollectEA`. As an implementation of `CollectEA`, we used EAT, the tool described above. As an implementation of `CoverageImpact`, we used a tool developed by the authors for use in earlier studies [10, 12]. (Note that these two implementations use the same underlying instrumentation tool, which reduces the internal threats to validity in our timing study.)

3.3 Study 1

In this study, we investigate RQ1. To evaluate relative execution costs for `CollectEA` and `CoverageImpact`, we measured the time required to execute an instrumented program on a set of test cases, gather the dynamic data (F and L arrays for `CollectEA`, and method coverage for `CoverageImpact`), and output that information to disk. We compare the execution costs for the two techniques to each other and to the cost of executing a non-instrumented program on the same set of test cases. Because we collected timing data for each individual test case, we computed our results by considering each test case in a test suite independently and then averaging the results across all test cases in the test suite.

The results of this study are shown in Table 4. For each program and version, the table reports the average execution time of each individual test case on the uninstrumented program, on the program instrumented by `CoverageImpact`, and on the program instrumented by `CollectEA`. It also reports the minimum, average, and maximum percentage overhead imposed by `CoverageImpact` (*%CoverageImpact Overhead*) and by `CollectEA` (*%CollectEA Overhead*).

As the table shows, the overhead imposed by `CollectEA` varies widely depending on the subject (on average about 110% for SIENA and 13% for JABA) and also for different executions of a given program version (e.g., it varies from 3% to 20% for JABA-LONG-v9). The overhead for `CoverageImpact` shows a similar trend.

After examining the results in more detail, we discovered that the observed variation is caused by a fixed cost associated with the instrumentation. Such fixed cost is due to the time required to (1) load and initialize the instrumentation-related classes and data structures, and (2) store the dynamic information on disk on program termination. For short running executions, such as the executions of SIENA, the fixed cost is considerable, whereas for longer executions is less relevant.
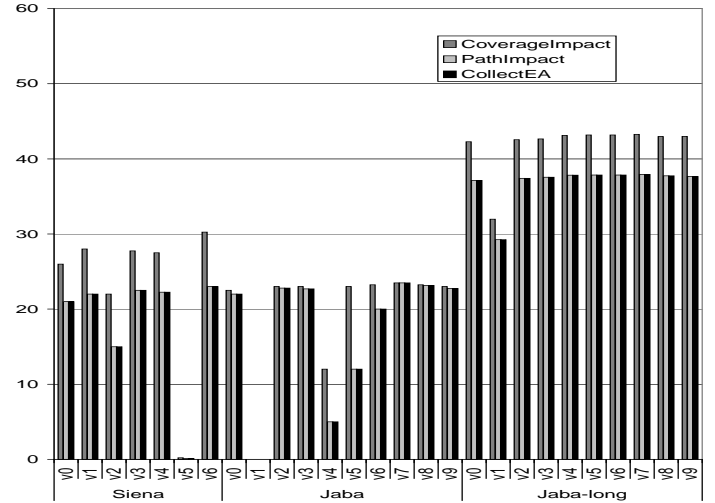

Figure 4: Precision results, expressed as percentage of methods in the impact sets.

For example, for JABA, we observed that all the executions that require more than a few seconds (about four seconds, for the executions considered) have an overhead consistently below 15% and as low as 3% in many cases. Although we do not have enough data points to generalize these results, they are encouraging. The results are especially encouraging if we consider that most real programs execute for more than a few seconds (e.g., most interactive programs). Moreover, we are using prototype, unoptimized tools, so it may be possible to reduce both fixed and variable costs associated with the instrumentation considerably.

Compared to `CoverageImpact`, `CollectEA` is, as expected, more expensive than `CoverageImpact`. However, the practical difference between the two techniques is small, ranging, on average, from 7% in the worst case (for *Jaba-long-v9*), to 3% in the best case (for *Jaba-long-v9*).

Therefore, we can conclude that `CollectEA` is practical for most programs, especially programs that run for more than a very short period of time. We can also conclude that `CollectEA` is applicable in all cases in which `CoverageImpact` is applicable.

3.4 Study 2

In this study, we investigate RQ2. To this end, we compare `CollectEA` with `CoverageImpact` in terms of precision. As a sanity check for our implementation, and to reduce the threats to internal validity, we also compare our technique with `PathImpact` to make sure that they produce the same results. To evaluate the precision of the techniques, we measure the relative sizes of the impact sets computed by the techniques on a given program, change set, and set of program executions. We report and compare such sizes in relative terms, as a percentage over the total number of methods.

This study is an extension of the studies presented in our previous work [11], in which we only considered a subset of the executions considered here (due to the cost of the most expensive technique considered, `PathImpact`). For this study, we implemented a version of technique `PathImpact` that does not compress the traces and, thus, has an acceptable time overhead (at the cost of a huge space overhead).

Program	Uninstrumented	CoverageImpact	CollectEA	%CoverageImpact Overhead			%CollectEA Overhead		
				min	avg	max	min	avg	max
Siena-v0	52	107	109	92	106	196	98	111	163
Siena-v1	52	107	109	93	106	204	98	111	166
Siena-v2	52	107	110	93	106	170	98	112	165
Siena-v3	53	108	110	91	104	189	98	108	164
Siena-v4	53	108	110	91	104	183	96	108	158
Siena-v5	53	108	110	80	104	194	88	108	166
Siena-v6	53	108	110	84	104	200	90	108	166
Jaba-v0	421	451	475	5	7	10	10	13	17
Jaba-v1	423	453	476	5	7	10	10	13	15
Jaba-v2	423	453	476	4	7	10	9	13	15
Jaba-v3	424	454	477	5	7	10	10	13	15
Jaba-v4	428	459	483	5	7	11	11	13	14
Jaba-v5	429	459	483	5	7	10	10	13	15
Jaba-v6	429	459	483	5	7	10	11	13	15
Jaba-v7	429	459	483	5	7	10	11	13	15
Jaba-v8	452	489	511	5	7	8	6	12	14
Jaba-v9	461	496	514	5	8	14	5	12	14
Jaba-long-v0	5170	5591	5728	3	9	15	3	12	27
Jaba-long-v1	5125	5497	5749	3	9	14	3	13	26
Jaba-long-v2	5128	5496	5737	1	8	11	3	13	26
Jaba-long-v3	5170	5508	5763	2	8	13	3	13	28
Jaba-long-v4	5300	5668	5903	2	8	13	3	13	26
Jaba-long-v5	5304	5641	5928	2	8	15	3	13	26
Jaba-long-v6	5363	5715	5960	1	8	15	3	13	30
Jaba-long-v7	5313	5693	5932	1	8	13	3	13	29
Jaba-long-v8	5338	5674	5943	1	7	12	5	12	20
Jaba-long-v9	5360	5689	5969	1	6	12	3	13	20

Table 4: Execution time (ms)

The graph in Figure 4 shows the results of the study. In the graph, each version of the two programs[6] occupies a position along the horizontal axis, and the relative impact-set size for that program and version is represented by a vertical bar—dark grey for technique CoverageImpact, light grey for PathImpact, and black for CollectEA. The height of the bars represents the impact set size, averaged across all test cases, expressed as a percentage of the total number of methods in the program.

As expected, the graph shows that, in all cases, the impact sets computed by CollectEA and PathImpact are identical (but computed at very different costs, as further discussed in Section 4).

The graph also shows that the impact sets computed by CollectEA are always more precise than those computed by CoverageImpact. In some cases, such differences in precision are considerable (e.g., for *Siena-v6*, *Jaba-v4*, and *Jaba-v5*). Therefore, the limited additional overhead imposed by CollectEA over CoverageImpact justifies its use.

4. RELATED WORK

We discussed existing dynamic impact analysis techniques in Section 1.2. We proved that our technique is as precise as PathImpact in Section 2.2 and showed that empirically in our studies (Section 3). In our studies, we also compared our technique with CoverageImpact in terms of precision and efficiency. In this section, we will present only a cost comparison between our technique and the others.

The space and time costs of technique CoverageImpact [10] are comparable to the corresponding costs for our technique.

---

[6]Note that we have two entries for each version of JABA because of the separation between short and long tests.

In terms of space, CoverageImpact collects, for each execution, one bit per method, whereas our technique collects, for each execution, two integers per method. In terms of time, the cost of the instrumentation required by the two techniques is similar, as shown in Section 3.

PathImpact is orders of magnitude more expensive than our technique because the cost of our technique is comparable to CoverageImpact[11]. In terms of time, PathImpact incurs significant runtime overhead because it compresses traces on the fly. In terms of space, the traces, even when compressed, can be very large, whereas our technique only requires two integers per method.

Breech and colleagues present an algorithm for computing the same impact sets as PathImpact on the fly [3]. Their algorithm collects, for each execution, an impact set for each method. At the entry of a method X, the algorithm adds X to the impact set of each method currently on the call stack. Then, it adds all methods on the call stack to X's impact set. This approach is less efficient than our approach both in terms of space and time. The worst-case space complexity is quadratic in the number of methods. The worst-case time complexity per method call is $O(n)$, where n is the number of methods, compared to $O(1)$ for our technique. Moreover, their technique assumes that all executions terminate with an empty call stack.

5. CONCLUSIONS

In this paper, we presented our new dynamic impact analysis technique. The technique is based on a novel algorithm that collects and analyzes only a small amount of dynamic information and is, thus, time and space efficient. We also presented a set of empirical studies in which we compare

our new technique with two existing techniques, in terms of performance and precision. The studies show that our technique is almost as efficient as the most efficient existing technique and is as precise as the most precise existing technique.

One important contribution of the paper is the definition of the Execute-After (EA) relation along with an efficient way to derive it. Although we presented and applied it in a specific context, the EA relation can be generalized along at least three dimensions. First, it can be generalized with respect to the level of granularity at which it is defined. We defined the relation at the method level, but it could be defined analogously at different levels (e.g., the basic-block level) or for different entities (e.g., entities for which the order of execution is important). Second, it can be generalized with respect to the programming language considered. Although we only described how to implement the technique for the Java language, it can be easily extended to other languages. Third, it can be generalized with respect to the multi-threaded programming as described in Section 2.3. Moreover, the concept of execute-after itself is more general than its specific meaning for impact analysis. Other dynamic analyses may be able to leverage the information provided by the EA relation.

We are considering several directions for future research. First, we will investigate dynamic impact analysis at finer granularity than the method level. In particular, we are currently implementing our technique at the statement level. Using that implementation, we will assess whether the gain in precision of the analysis at the statement level justifies the additional cost. Second, we will continue our previous work on performing analysis of deployed software [13] by applying the new technique presented in this paper to real software released to real users. The additional precision of this technique will let us better assess differences between in-house and in-the-field behaviors [10]. Finally, we will investigate ways to incorporate static analysis into the presented technique, to further improve its efficiency and precision. For example, we could use static analysis to identify pairs of methods whose relative execution order can be determined statically and use that information to reduce the amount of instrumentation inserted in the program under analysis. For another example, static slicing may let us further reduce the size of the dynamic impact sets by eliminating parts of the program that are executed after the changed parts, but are not dependent on the changes.

6. ACKNOWLEDGEMENT

7. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, September 2004.

[2] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Press, Los Alamitos, CA, USA, 1996.

[3] B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of the International Conference of Software Maintenance*, September 2004.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computing Systems*, 19(3):332–383, August 2001.

[5] A. Chawla and A. Orso. A generic instrumentation framework for collecting dynamic information. In *Online Proceeding of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004)*, july 2004.

[6] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 430–441, Nov. 2003.

[7] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the International Conference on Software Engineering*, pages 308–318, May 2003.

[8] J. L. Lions. ARIANE 5, Flight 501 Failure, Report by the Inquiry Board. *European Space Agency*, July 1996.

[9] J. P. Loyall, S. A. Mathisen, and C. P. Satterthwaite. Impact analysis and change management for avionics software. In *Proceedings of the IEEE National Aeronautics and Electronics Conference, Part 2*, pages 740–747, July 1997.

[10] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 128–137, September 2003.

[11] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the International Conference on Software Engineering*, pages 491–500, May 2004.

[12] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metadata to support the regression testing of component-based software. In *Proceedings of the International Conference on Software Maintenance*, pages 716–725, November 2001.

[13] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 65–69, Rome, Italy, July 2002.

[14] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, Englewood Cliffs, NJ, 1998.

[15] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis for java programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, October 2004.

[16] R. J. Turver and M. Munro. Early impact analysis technique for software maintenance. *Journal of Software Maintenance*, 6(1):35–52, Jan. 1994.