Refining Interprocedural Change-Impact Analysis using Equivalence Relations

Alex Gyori
University of Illinois, USA
gyori@illinois.edu

Shuvendu K. Lahiri
Microsoft Research, USA
shuvendu@microsoft.com

Nimrod Partush
Technion, Israel
nimi@cs.technion.ac.il

ABSTRACT
Change-impact analysis (CIA) is the task of determining the set of program elements impacted by a program change. Precise CIA has great potential to avoid expensive testing and code reviews for (parts of) changes that are refactorings (semantics-preserving). However, most statement-level CIA techniques suffer from imprecision as they do not incorporate the semantics of the change.

We formalize change impact in terms of the trace semantics of two program versions. We show how to leverage equivalence relations to make dataflow-based CIA aware of the change semantics, thereby improving precision in the presence of semantics-preserving changes. We propose an anytime algorithm that applies costly equivalence-relation inference incrementally to refine the set of impacted statements. We implemented a prototype and evaluated it on 322 real-world changes from open-source projects and benchmark programs used by prior research. The evaluation results show an average 35% improvement in the number of impacted statements compared to prior dataflow-based techniques.

CCS CONCEPTS
•Software and its engineering → Automated static analysis; Software verification;

KEYWORDS
Impact Analysis, Software Maintenance, Equivalence

1 INTRODUCTION
Software constantly evolves to add and improve features, eliminate bugs, improve design, etc. As software evolves faster than ever, it requires rigorous techniques to ensure that changes do not modify existing behavior in unintended ways. Some of the emerging approaches to ensure the quality of a change are code reviews [35], regression testing [19, 44], test-suite augmentation [34, 39, 40], code contracts [5, 25], regression verification [20, 38] and verification modulo versions [33]; all benefit from change-impact analysis (CIA).

Change-impact analysis determines the set of program elements that may be impacted by a syntactic change. Traditional approaches are coarse-grained and operate at the level of types and classes [1, 2], or files [19] to retain soundness. Fine-grained techniques that aim to work at the level of statements are typically based on performing dataflow analysis [43] on one program to propagate the change along data and control flow edges [3, 10, 32]. Such techniques fail to take the semantics of the change into account; therefore, they cannot distinguish between changes that a user expects to have only local impact on existing code (e.g., a code refactoring) from ones that have substantial impact on existing code (e.g., changing the functionality or fixing a bug). The ability to distinguish changes whose impact is local (limited to the changed procedure or a few callers or callees within one or two levels) can help with code review and regression-testing efforts. Changes with substantial impact can be prioritized for more rigorous code reviews and more testing.

In this paper, we aim to improve the precision of CIA by leveraging equivalence relations between the variables of two programs across a change. At a high level, these equivalences help prune the flow of a change along the data or control flow edges of the changed program. To integrate such equivalences, we first formalize the notion of change impact precisely in terms of the trace semantics of two programs. Next, we show how to make CIA change-semantics aware by incorporating various equivalence relations into an interprocedural dataflow analysis. Since computing equivalence relations is expensive, we propose an anytime algorithm [46, 48] to incrementally compute equivalence relations.

1.1 Overview
Figure 1 shows a running example in C. The example is inspired by real commits to Coreutils, in files paste.c [13] and sort.c [14].

The program has three changes. Two are semantics-preserving: (i) extracting the literal ‘\n’ into the variable line_delim in the procedure print_product_info (lines 1, 4, 5) and (ii) replacing the conditional operator with a double negation in locale_ok (lines 22, 23)\(^1\). The third change is not semantics-preserving: it sets the line_delim variable to ‘\0’ (a different value than in the old version) in the procedure print_product_info (lines 12, 14, 14), which impacts statements in print_minors.vers. We claim the only (syntactically unchanged) line that is impacted by the changes is the highlighted line 41 (assume for this example that all executions start from the procedure print_product_info); a statement is impacted, intuitively, if the sequence of values it reads can differ when executing the two versions of the program in the same environment. For brevity, we omit the definitions of the

\(^1\)Negation in C coerces the values to 0 or 1.
has a change to its argument that marks all the statements in the procedure as impacted because they all depend on the changed argument. Next, the call to locale_ok on line 6 impacts the locale variable because of the change to the body of locale_ok and the data dependency of the return value on the change. This in turn will mark the input of print_name as impacted at line 8, which in turn flows to its output because the return value is control dependent on the input variable marked as impacted (a context-insensitive analysis will impact the return at all call sites to print_name). This impact through the return value will propagate to the call to print_majorvers and print_minorvers because of the control dependency on prin and will impact all the statements in these procedures as well as all the returns at both call sites. Finally, the call to print_minorvers will impact all of the callee statements. A context-sensitive analysis does not help either because the body of locale_ok changes, which implies that the return value may change across the two versions. This is sound but imprecise since the analysis is unable to determine that the statements in print_name and print_majorvers are not impacted.

Equivalence: A traditional interprocedural equivalence checking [20, 28] (checking if two procedures have identical input-output behavior) will find that locale_ok, print_name, print_header, print_majorvers, and print_minorvers have identical summaries. This is unsound for the question of impact analysis, as the statement of print_minorvers is impacted due to the change of print delimiter. This illustrates the difference between CIA and (traditional) equivalence checking: two procedures can be equivalent, but still impacted, because they may get called under different contexts and exhibit different behaviors.

Our approach: Our change-semantics aware CIA works as follows: it infers equivalence relations over variables and determines that the arguments at all call sites to print_name and print_majorvers are equal in both versions and stops propagating impacts through their arguments. Further, locale_ok has an equivalent summary in the two versions (by using equivalence checking)—this ensures that two call sites with equal arguments return equal results. From these two facts, the technique infers (by simple dataflow analysis) that arguments to print_name and print_majorvers are not impacted and therefore the statements in both print_name and print_majorvers are not impacted. Thus, our approach precisely identifies the only unchanged impacted line as line 41.

1.2 Contributions

In this work, we make the following contributions:

(1) We precisely formalize the set of statements impacted by a change, in terms of the trace semantics of two versions of a program (§ 3.1).
(2) We make a dataflow-based CIA change-semantics aware by incorporating various equivalence relations (§ 4).
(3) We describe an anytime algorithm that allows incrementally computing more equivalences to refine the analysis at the expense of time (§ 4.1).
(4) We have implemented a prototype using SymDiff [28, 29], and evaluated our technique on 322 real-world changes collected from GitHub open-source projects and several standard benchmark programs used in prior research [24].
2 BACKGROUND

For the ease of presentation, we will formalize the problem and our technique over a simple language. We can compile most constructs of general-purpose imperative programming languages to our simple language [4, 12, 18, 41]; we discuss this in § 2.2.

2.1 A Simple Language

A program consists of procedures represented as control-flow graphs, statements, and expressions.

**Expressions:** $e \in Exprs$ in the language are built up from constants, variables and operator applications:

$$e \in Exprs \equiv c \mid x \mid \ldots \mid op(e_1, \ldots, e_k)$$

Here $c$ represents *constant* values of different types such as \{true, false\} for Booleans, \{-1, 0, 1, \ldots\} for integers, and $x$ denotes variables in scope. An operator $op$ is a function or predicate symbol that can be uninterpreted or interpreted by some theories (e.g., $\{+, -, \times, \geq, \ldots\}$ by the theory of arithmetic). We represent a vector of variables and expressions using $\vec{x}$ and $\vec{f}$, respectively.

**Statements:** $st \in Stmts$ are comprised of assign, assume, skip and procedure call statements.

$$st \in Stmts \equiv x := e \mid \text{assume } e \mid \text{skip} \mid \text{call } x_1, x_2, \ldots, x_k \leftarrow f(e_1, e_2, \ldots, e_m)$$

The argument to *assume* is a Boolean-valued expression, and a *skip* is a no-op. A call statement can have multiple return values and they are assigned to variables $x_i$ at the call site.

**Procedures:** A procedure $f \in Procs$ is represented as a control-flow graph consisting of $(N_f, E_f, In_f, Out_f, Vars_f, n_f^1, n_f^2)$, where:

- $N_f$ is a set of control-flow locations in $f$,
- $E_f \subseteq N_f \times N_f$ is a set of edges over $N_f$ denoting control-flow,
- $In_f$ (respectively, $Out_f$) is the vector of input (respectively, output) forms of $f$. The output forms model return values and output parameters,
- $Vars_f$ is the set of variables in the scope of $f$ and includes $In_f$, $Out_f$, and local variables of $f$,
- $n_f^1 \in N_f$ (respectively, $n_f^2 \in N_f$) is the unique entry (respectively, exit) node of $f$.

Let $N = \bigcup_{f \in Procs} N_f$ and $Vars = \bigcup_{f \in Procs} Vars_f$. Nodes and variables in a procedure $f$ are often denoted by $n_f$ and $x_f$ respectively. For any node $n_f \in N_f$, we define the readset $RWars(n_f)$ and writeset $WVars(n_f)$ as the set of variables that are read and written to respectively in the statement at $n_f$.

A program $Prog \in Progs$ is a tuple $(Procs, main, StmtsAt)$ where:

(i) Procs is a set of procedures in the program,
(ii) main $\in Procs$ is the entry procedure from which the program execution starts, and
(iii) StmtsAt : $N \rightarrow Stmts$ maps a node $n \in N$ in a procedure $f$ to a statement. For any $f$, we assume that both $StmtsAt(n_f^1) = \text{skip}$ and $StmtsAt(n_f^2) = \text{skip}$.

2.2 Expressiveness

We can compile most constructs in general-purpose imperative programming languages to our simple language. This follows the same principle as translators from languages such as C and Java to the Boogie language [4, 12, 18, 41].

**Control flow:** Loops can be automatically transformed into tail-recursive procedures [20, 28, 29]. We use $n_1 : st_1 \text{ goto } n_2, n_3$ to express that $StmtsAt(n_1) = st$ and $(\{n_1, n_2\} \cup \{n_3\}) \subseteq Ef$. A conditional statement if ($e$) $st_1$ else $st_2$ is modeled as:

$$n_1 : x := e; \text{ goto } n_2, n_3;$$

$$n_2 : \text{ assume } x; st_1; \text{ goto } n_4; n_3 : \text{ assume } \neg x; st_2; \text{ goto } n_4;$$

where a fresh Boolean variable $x$ captures the value of the condition $e$.

We assume that each node $n \in Ef$ has at most two successor nodes in $Ef$ corresponding to conditional statements branches. The only use of an *assume* statement is to model a conditional statement. We refer to $n_1$ as a branching node with two successors in $Ef$ with complementary expressions in *assume* statements.

**Globals and heap:** Richer data types such as arrays and maps can be modeled, e.g., array read $x[e]$ is modeled using $sel(x, e)$ and a write $x[e] := e_2$ is modeled using $x := update(x, e_1, e_2)$ [6]. Arrays are in turn used to model the heap in imperative programs and are standard in most software verification tools [12, 18, 41]. Additional internal non-determinism (e.g., read from file, network) is lifted as reads from immutable input arrays of main, making programs deterministic in our language [28]. We desugar the program’s global variables (including the heap) as additional input and output formal arguments to a procedure. We transform each procedure into its Static Single Assignment (SSA) form [17], where a variable is assigned at exactly one program node.

2.3 Semantics

Let $\mathcal{V}$ denote the set of values that variables and expressions can evaluate to. Let $\theta \in \Theta$ be a store mapping variables to values in $\mathcal{V}$. For $x \in Vars$, we define $x \in \theta$ if $x$ is a variable in the domain of $\theta$. For $x \in \Theta$, $\Theta(x)$ denotes the value of variable $x$. The store $[x \rightarrow v]$ represents a singleton store that maps $x$ to $v$. The store $\theta|_{Vars_f}$ restricts the domain of the store to variables in $Vars_f$. For stores $\theta_1$ and $\theta_2$, the store $\theta_3 = \theta_1 \oplus \theta_2$ is defined as follows for any variable $x \in \theta_1$ or $x \in \theta_2$:

$$\theta_3(x) = \begin{cases} \theta_2(x), & \text{if } x \in \theta_2 \\ \theta_1(x), & \text{otherwise} \end{cases}$$

The value of an expression $e \in Exprs(\theta(e))$ is defined inductively on the structure of $e$ (we omit it for brevity as it is fairly standard).

**Calls:** Let $cs \in (N \times Vars^* \times \Theta)^*$ be a call stack that is a sequence of tuples $((n_0, \theta_0, \theta_0), (n_1, \theta_1, \theta_1), \ldots)$, where $n_i$ is the $i$-th call site on the call stack $(n_0)$ is the most recent), $\theta_i$ and $\theta_{i+1}$ respectively, are the vector of return actuals and the valuation of the local variables of the caller, at the corresponding call site. Let $CS$ denote the set of all such call stacks, $e$ denotes an empty stack, and $(n, \theta, \theta) : cs$ denotes the concatenation operator.

**Transition Relation:** A state $\sigma \in \Sigma$ is a tuple $(n, \theta, cs) \in N \times \Theta \times CS$ that denotes a point in program execution where $n$ is the current node being executed in a procedure $f$, $\theta$ is the valuation of variables in $Vars_f$ and $cs$ is the current call stack.

A state transition denoted as $(n_f, \theta_1, cs_1) \rightarrow (n_2, \theta_2, cs_2)$ is a relation over $\Sigma \times \Sigma$ that holds only if:

$\text{The introduction of } x \text{ simplifies determining if control flow is impacted by only inspecting the conditional node}$
A partial bijection is a partial function that is injective when defined and (trivially) surjective when restricted to its image [21].
### 3.3 Dataflow-Based Change-Impact Analysis

In this section, we describe Dataflow-based Change-Impact Analysis (DCIA), a change semantics unaware static analysis that provides a conservative estimate of the set of impacted nodes. The static analysis is an interprocedural dataflow analysis [43] that starts with a program $Prog^i$ ($i \in 1, 2$) and a conservative estimate of the syntactically-changed nodes, nodes not in Mapped, and returns an upper bound on the set of (a) impacted nodes, (b) impacted variables, and (c) output variables whose summary may have changed.

**Predicates:** Table 1 defines some straightforward predicates used in the inference rules. The $\text{OutputActual}(r, i, f, n)$ predicate holds when the $i^{th}$ return value is assigned to variable $r$, at the call to $f$ from the node $n$ (note that we allow multiple return values); we call $r$ the output actual to differentiate it from the $i^{th}$ output formal inside the callee. For $\text{ControlDependent}(n_2, n_1)$, a node $n_2$ is control-dependent on node $n_1$ iff (i) there exists a path from $n_1$ to $n_2$ s.t. every node in the path other than $n_1$ and $n_2$ is post-dominated by $n_2$, and (ii) $n_1$ is not post-dominated by $n_2$ [17].

**Dependency:** Figure 2 describes a set of inference rules to compute two relations $\text{DependsOnVar}$ and $\text{DependsOnNode}$. For a pair of variables $x, y \in \text{Vars}$ such that $y$ is either data- or control-dependent on $x$, then $\text{DependsOnVar}(y, x, f)$ holds. Similarly, a node $n \in \text{N}_f$ and a variable $x$ that is updated at $n$, $\text{DependsOnNode}(x, n, f)$ holds. Subsequently, any variable $y$ such that $y$ is data or control dependent on such a variable $x$, then $\text{DependsOnNode}(y, x, f)$ holds. An inference rule (e.g. $\text{DependsNode}$) lists a set of antecedents (above the line) and the consequent (below the line). Applying an inference rule results in adding a tuple to the relation in the consequent (e.g. $\text{DependsOnNode}$). The inference rules are applied repeatedly until a fix-point is reached.

Most of the inference rules are straightforward encoding of program data- and control flow. The rule $\text{ControlDepends}$ expresses that if $n_1$ is a branching node, whose condition depends on $x$ and $y$ is written in a control-dependent node $n_2$, then $y$ depends on $x$. The rule $\text{SummaryDepends}$ captures the dependency of an actual return $r$ on a variable $w$ passed as an argument to $f$ in a caller $g$, where $w$ indirectly flows to $r$ through a procedure call to $f$. For this call site, the $i$-th output formal $y$ (which is assigned to the output actual $r$) is dependent on the $j$-th input formal $x$, which in turn is assigned the actual $e$ at the call site.

---

**Table 1: Predicates used for dataflow analysis.**

<table>
<thead>
<tr>
<th>Predicate name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{BranchingNode}(n)$</td>
<td>if $n$ is a branching node</td>
</tr>
<tr>
<td>$\text{ControlDependent}(n_2, n_1)$</td>
<td>if $n_2$ is control-dependent on $n_1$ [17]</td>
</tr>
<tr>
<td>$\text{Callsite}(n, f, g)$</td>
<td>if $\text{StmtAt}(n)$ is a call to $f$ within a caller $g$.</td>
</tr>
<tr>
<td>$\text{InputFormal}(x, i, f)$</td>
<td>if $x$ is the $i$-th input formal of $f$</td>
</tr>
<tr>
<td>$\text{OutputFormal}(x, i, f)$</td>
<td>if $x$ is the $i$-th output formal of $f$</td>
</tr>
<tr>
<td>$\text{InputActual}(e, i, f, n)$</td>
<td>if the expression $e$ is the $i$-th actual argument to a call to $f$ at a call site $n$</td>
</tr>
<tr>
<td>$\text{OutputActual}(r, i, f, n)$</td>
<td>if the variable $r$ receives the $i$-th output formal to a call to $f$ at a call site $n$</td>
</tr>
</tbody>
</table>

---

**Impact Analysis:** Figure 3 describes a set of inference rules to compute the set of nodes that are impacted in either program. For now, we ignore the highlighted antecedents (we use them in § 4 where we describe how we incorporate change semantics). The rules take as input a program (either $Prog^1$ or $Prog^2$), the set of mapped nodes Mapped, and precomputed relations $\text{DependsOnNode}$ and $\text{DependsOnVar}$ for the particular program. They produce the relations $\text{ImpactedNode}$, $\text{ImpactedVar}$, and $\text{ImpactedSumm}$ that are an upper bound on the set of impacted nodes, variables, and variable summaries, respectively. Next we explain the rules using our illustrative example.

The SYNT-CHANGED rule represents the source of any change impact, stemming from syntactic changes to the program. The next rules, prefixed by VAR-2 and NODE-2, propagate impact from variables to expressions or program nodes, and vice versa.

The next two rules propagate change impact for an output $y$ of a procedure $f$, expressed by the $\text{ImpactedSumm}(y, f)$ predicate. For an output formal $y \in \text{Out}_f$, the summary (input-output dependency) may change either when (i) $y$ depends on a variable updated at an unmapped node $n \in \text{N}_f$ (expressed by $\text{IMPACT-SUMMARY}$), or (ii) $y$ depends on the return of a procedure $g$ with an impacted summary (expressed by $\text{IMPACT-SUMMARY-PROP}$).

Next, the CALL-IMPACT rule says that an input formal $x$ in $f$ can be impacted if the corresponding actual argument $e$ at a call site is impacted, representing downward flowing impact where a caller impacting a callee. Alternatively, the RETURN-IMPACT rule considers the case when the variable summary for the corresponding output formal $y$ is impacted, representing upward flowing impact.
Finally, SUMMARY-IMPACT considers impact which propagates through a callee. Here, g calls f with an impacted actual e for the formal input x of f. Since the formal output y of f depends on aforementioned impacted x, the impact flows back outwards into the output actual r in g.

Our analysis preserves context-sensitivity as it does not impact a return value simply because the corresponding output formal is impacted in some context.

The algorithm DCIA does the following:

1. Takes as input $\text{Prog}^1, \text{Prog}^2$ and MAPPED.
2. Applies the inference rules in Figure 3 on $\text{Prog}^2$ to generate $\text{IMPACT-Node}^1, \text{IMPACT-Var}^1, \text{IMPACT-Summ}^1$ until a fix-point is reached.
3. Returns the tuple $(t)\text{IMPACT-Node}^1, \cup t\text{IMPACT-Var}^1, \cup t\text{IMPACT-Summ}^1$.

The following theorem states the soundness of the dataflow analysis DCIA.

**Theorem 3.2 (Soundness).** Given two programs $\text{Prog}^1, \text{Prog}^2 \in \text{Programs and MAPPED} \subseteq N$, (a) DCIA terminates, and (b) for any $n \not\in \text{IMPACT-Node}$, n is not an impacted node with respect to MAPPED (according to Definition 3.1).

Consider for example the changes in Figure 1 at line 22; the procedure `locale.ok` has an impacted summary because its return variable depends on a node that is syntactically changed, i.e., is not in MAPPED. This causes the line 6 and the variable `locale` to be marked as impacted because of the rule IMPACT-SUMMARY. Impacts are propagated interprocedurally by the rule CALL-IMPACT to all calls that take `locale` as an argument, i.e., `print_name, print_major_vers, and print_minor_vers`. Similarly, using the same rule, the body of `print_header` is impacted by the changed argument `\"n\"` changed to the variable `line_delim` on line 4. The propagation through calls further impacts their entire body because of the data and control dependency on the impacted argument (by the rules NODE-2-VAR and VAR-2-NODE which propagate impact through both control- and data-dependency relying on the predicate $\text{DEPENDS-ON-Var}$).

## 4 INCORPORATING CHANGE SEMANTICS

In this section, we make the DCIA algorithm change-semantics aware. In other words, the analysis takes into account also the exact semantics of the change, in addition to the set of nodes MAPPED that may have been syntactically changed. We inject the change-semantics by leveraging equivalence relationships between variables and procedure summaries in the two programs $\text{Prog}^1$ and $\text{Prog}^2$.

Let us define the following semantic equivalences for a variable over $\text{Prog}^1$ and $\text{Prog}^2$.

**Definition 4.1 ($\text{Pre-Equiv}$).** $\text{Pre-Equiv}(x, f)$ holds for an input formal $x \in \text{Inf}$ if for all stores $\theta$ and for every pair of traces $t^1 \leq t^1_{\text{main}}(\theta)$ and $t^2 \leq t^2_{\text{main}}(\theta)$, $(\tau^1 l_{\text{f}})^x = (\tau^2 l_{\text{f}})^{\text{f} \theta} l_{\text{f}}^x$.

Intuitively, $\text{Pre-Equiv}(x, f)$ holds for an input formal $x$ of $f$ if any two executions starting from $\text{main}$ on the same input $\theta$ call $f$ with the same sequence of values of $x$. For the example in Figure 1 the equivalences that hold are $\text{Pre-Equiv}($delim, print_header$)$, $\text{Pre-Equiv}($locale, print_name$)$, $\text{Pre-Equiv}($locale, print_major_vers$)$, and $\text{Pre-Equiv}($locale, print_minor_vers$)$. In contrast, the equivalence $\text{Pre-Equiv}($delim, print_minor_vers$)$ does not hold, because of different values for $\text{delim}$ ‘\n’ and ‘\’ respectively, at the call-site in $\text{print_product_info}$.

We define $\text{Dep}(y)$ as the set of variables $x$ in either $\text{Prog}^1$ or $\text{Prog}^2$ such that $\text{DEPENDS-ON-Var}(x, y, f)$. For two stores $\theta_1$ and $\theta_2$
defined over same set of variables, we denote $\theta_1 = Vars_1$, $\theta_2$ to mean $\theta_1(x) = \theta_2(x)$ for every $x \in Vars_1$.

Definition 4.2 (SummaryEquiv). $\text{SummaryEquiv}(y, f)$ holds for an output formal $y \in \text{Out}_f$ if $(\theta_1, \theta_2) \in \Omega_f$ in $\text{Prog}^2$ and $\theta_1 = \text{Dep}(y) \theta_3$, then $(\theta_3, \theta_4) \in \Omega_f$ in $\text{Prog}^2$ ($i \neq i$) and $\theta_2(y) = \theta_4(y)$.

Intuitively, if the versions of $f$ are executed from stores $\theta_1$ and $\theta_2$ where $\theta_1 = \text{Dep}(y) \theta_3$, then either both procedures do not terminate, or the value of $y$ after executing $f$ is identical on exit. In Figure 1, all procedures are equivalent except $\text{pr\_int\_product\_info}$, i.e., in this case $\text{SummaryEquiv(line\_delim\_print\_product\_info)}$ does not hold since in one version the value of $\text{line\_delim}$ at the end of the execution is "$\emptyset$" while in the other it is undefined.

Figure 3 with the highlighted parts provides a refinement to the dataflow analysis to incorporate change semantics. In addition to the $\text{MAPPED}$, the algorithm now takes as input pre-computed relations $\text{PreEquiv}$ and $\text{SummaryEquiv}$. In this section, we assume an oracle that provides these relations; we provide one implementation later (§ 5.1). The highlighted facts strengthen the antecedent of a rule and prevent it from being applicable in some contexts. For example, the strengthened $\text{CALL\_IMPACT}$ prevents an input formal $x$ from being impacted if $\text{PreEquiv}(x, f)$ holds. Similarly, the strengthened $\text{IMPACT\_SUMMARY}$ prevents a summary for $y$ from impact if we know that $\text{SummaryEquiv}(y, f)$ holds. The strengthened $\text{SUMMARY\_IMPACT}$ is now applicable only when either (i) the formal $x$ does not satisfy $\text{PreEquiv}$ or (ii) the summary for $y$ does not satisfy $\text{SummaryEquiv}$.

We denote the new change-semantics aware algorithm as Semantic Dataflow-based Changed Impact Analysis (SEM-DCIA).

Theorem 4.3 (Soundness). Given two programs $\text{Prog}^1$, $\text{Prog}^2$ in Programs, $\text{MAPPED}$, $\text{PreEquiv}$, and $\text{SummaryEquiv}$, (i) SEM-DCIA terminates, and (ii) for any $n \notin \text{ImpactedNode}$, $n$ is not an impacted node with respect to $\text{MAPPED}$ (from Definition 3.1).

4.1 Anytime Algorithm

The SEM-DCIA algorithm assumes an oracle to compute the $\text{PreEquiv}$ and $\text{SummaryEquiv}$ relations. Computing such equalities typically require constructing the product of the two programs $\text{Prog}^1$ and $\text{Prog}^2$ and inferring equivalence relations over the product program [29]. Such inference algorithms typically have high complexity and therefore it is wise to apply them prudently. In this section, we make a simple observation that allows us to interleave SEM-DCIA and inference of $\text{PreEquiv}$ and $\text{SummaryEquiv}$ in a single framework.

Algorithm 1 (SEM-DCIA-ANYTIME) provides an anytime algorithm that performs the integration. The algorithm takes as an additional input $\text{Procs}^A$, the set of syntactically changed procedures. It outputs a set of nodes $\text{impNds}$ that overapproximates the set of impacted nodes. We term the algorithm anytime [15, 46, 48] because the algorithm can be stopped at any time after the first call to SEM-DCIA to obtain a conservative bound for the impacted nodes.
The algorithm starts with invoking SEM–DCIA on the two programs with an empty set of equivalences in EQ (line 4); this is identical to calling DCIA. The return values provide a conservative measure on impacted variables, nodes and summaries respectively (Theorem 3.2). The algorithm implements a loop (line 6) where it increases the frontier of procedures Proc′ among Proc's that are analyzed for inferring equivalences in InferEquivs (line 13). Lines 7 and 8 construct equivalences from the provably non-impacted variables and summaries. These equivalences are added to EQ in line 9. ProcsWithin returns all procedures that can reach or be reached from Proc′ within a call stack of depth k; k is incremented with each iteration of the loop. AbstractProc's abstracts all procedures outside Proc′; it only retains the knowledge of whether any procedure f ∈ Proc′ has additional call sites outside Proc′ - this determines whether PreEquiv can be inferred for a procedure. InferEquivs is invoked with a set of equivalences in EQ on the smaller programs Prog′. The final call to SEM–DCIA is used to compute the more refined set of impacted variables, nodes and summaries based on the equivalences discovered from InferEquivs. The loop terminates when Proc′ consists of the entire program; at this point InferEquivs has already looked at the entire program and no new equivalences will be discovered in line 13.

Let us denote SEM–DCIAk as an instantiation of the algorithm SEM–DCIA–ANYTIME that terminated after the loop is executed exactly k + 1 times. We also denote SEM–DCIA∞ if the loop terminates normally after Proc′ equals Proc.

Theorem 4.4 (Soundness). Given two programs Prog1, Prog2 ∈ Programs, MAPPED, and Proc′, if SEM–DCIAk terminates then for any n ∈ impNd's, n is not an impacted node with respect to MAPPED (according to Definition 3.1).

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We presented and evaluated our SEM–DCIA() analysis for C programs, but our analysis is implemented over the intermediate verification language Boogie [4]. We leverage SMACK [41] to convert LLVM bytecode to Boogie programs.

Differencing: For our initial implementation, we leveraged diff over C files to produce the list of changes, i.e., nodes not in MAPPED. However, diff does not satisfy the soundness criteria for diff (see Section 3.1) because of changes in macros, data structures, control-flow changes, etc.; we therefore conservatively consider all nodes in a changed procedure as sources of impacts. Note that because we operate on Boogie, macros are already expanded so changes in macros will be reflected in the resulting Boogie code. Although this can overapproximate the initial source of impact, the use of equivalences in SEM–DCIA allows us to prune the spurious impacts from escaping the syntactically-changed procedures; All our code and scripts are available in the SymDiff repository at: https://symdiff.codeplex.com/.

Inference: We used SymDiff to construct a product program and infer valid PreEquiv and SummaryEquiv. Given Prog1 and Prog2, SymDiff generates a product program Prog1 × Prog2 that defines a procedure f × π for every f and π( f) ∈ Prog3. For the product program Prog1 × Prog2, one can leverage any of the (single program) invariant generation techniques to infer preconditions, postconditions (including two-state postconditions) on f × π. Such invariants are relational in that they are over the state of two programs Prog1 and Prog2, and include equivalences relations such as PreEquiv (preconditions of f × π) and SummaryEquiv (summary of f × π). To ensure our inferred equivalences are valid we require the programs to be equi-terminating [23]; this is an area of future work – for now we assume that changes do not introduce non-termination. We modified SymDiff to add candidates for inferring summaries and take as input cheaply-inferred equivalences from DCIA. More details can be found in our extended report [22].

5.2 Evaluation

In this section we evaluate the effectiveness of our approach on GitHub projects with real program changes and standard benchmark programs with artificial changes. We show that our semantic based analysis, SEM–DCIA improves on DCIA by reducing the size of the impacted set, a proxy metric for the effort necessary to perform many software engineering tasks such as code review and testing.

We analyze 164 changes consisting of refactorings, feature additions, buggy changes, and bug fixes from 5 GitHub projects. We selected the projects based on popularity, size, active development, and compatibility with SMACK. The projects, number of versions used, their size in non-comment non-blank source lines of code (SLOC), and corresponding change sizes (in number of C source lines changed) are summarized in Table 2. Our subjects are C implementations of a virtual machine program (tinyvm), a histogram creator (histo), a markdown presentation tool (mdp), a file-descriptor management library (flingfd) and a test-generation library (theft). We include 6 standard benchmarks widely used by prior research [24]. These benchmarks consist of 158 manually introduced changes representing non-trivial and hard to detect bugs. Our projects are sized between 142 lines of source code and 6205 (SLOC). The changes in our projects vary in size between very small changes, consisting of single line changes and larger ones, consisting of over 400 lines (most of our changes are small).

For our experiments, we first compare SEM–DCIA against DCIA to study the impact of adding change-semantics to the impact analysis (§ 5.3). Next, we evaluate the cost-precision tradeoff of the anytime algorithm SEM–DCIA–ANYTIME (§ 5.4). Finally, we present several representative examples discovered while applying our tool (§ 5.5).

Table 2: Summary of projects used as evaluation subjects

<table>
<thead>
<tr>
<th>Project Name</th>
<th># Version Pairs</th>
<th>SLOC min</th>
<th>SLOC max</th>
<th>LOC Changed min</th>
<th>LOC Changed max</th>
</tr>
</thead>
<tbody>
<tr>
<td>flingfd</td>
<td>2</td>
<td>142</td>
<td>146</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>histo</td>
<td>8</td>
<td>617</td>
<td>624</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>mdp</td>
<td>91</td>
<td>135</td>
<td>1616</td>
<td>1</td>
<td>402</td>
</tr>
<tr>
<td>theft</td>
<td>2</td>
<td>1672</td>
<td>1858</td>
<td>2</td>
<td>328</td>
</tr>
<tr>
<td>tinyvm</td>
<td>61</td>
<td>425</td>
<td>903</td>
<td>1</td>
<td>328</td>
</tr>
<tr>
<td>print_tokens</td>
<td>5</td>
<td>478</td>
<td>480</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>397</td>
<td>402</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>replace</td>
<td>32</td>
<td>509</td>
<td>516</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>schedule</td>
<td>9</td>
<td>290</td>
<td>294</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>space</td>
<td>38</td>
<td>6180</td>
<td>6205</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>tcas</td>
<td>41</td>
<td>136</td>
<td>140</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>tot_info</td>
<td>23</td>
<td>346</td>
<td>347</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Table 3: Analysis results for different levels of precision. Time in seconds. (timeout = 1 hour)

<table>
<thead>
<tr>
<th>Project Name</th>
<th>DCIA</th>
<th>SEN-DCIA&lt;sub&gt;0&lt;/sub&gt;</th>
<th>SEN-DCIA&lt;sub&gt;1&lt;/sub&gt;</th>
<th>SEN-DCIA&lt;sub&gt;∞&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
<td>Time</td>
<td>min</td>
</tr>
<tr>
<td>flingfd</td>
<td>64</td>
<td>84</td>
<td>0.94</td>
<td>39</td>
</tr>
<tr>
<td>histo</td>
<td>0</td>
<td>86</td>
<td>2.14</td>
<td>0</td>
</tr>
<tr>
<td>mdp</td>
<td>0</td>
<td>465</td>
<td>28.16</td>
<td>0</td>
</tr>
<tr>
<td>tinyvm</td>
<td>0</td>
<td>344</td>
<td>68.96</td>
<td>0</td>
</tr>
<tr>
<td>theft</td>
<td>184</td>
<td>261</td>
<td>4.48</td>
<td>11</td>
</tr>
<tr>
<td>print_tokens</td>
<td>151</td>
<td>153</td>
<td>2.22</td>
<td>69</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>155</td>
<td>158</td>
<td>1.46</td>
<td>80</td>
</tr>
<tr>
<td>replace</td>
<td>75</td>
<td>195</td>
<td>4.96</td>
<td>74</td>
</tr>
<tr>
<td>schedule</td>
<td>79</td>
<td>115</td>
<td>1.37</td>
<td>7</td>
</tr>
<tr>
<td>space</td>
<td>20</td>
<td>2851</td>
<td>59.45</td>
<td>14</td>
</tr>
<tr>
<td>tcas</td>
<td>1</td>
<td>49</td>
<td>0.66</td>
<td>0</td>
</tr>
<tr>
<td>tot_info</td>
<td>103</td>
<td>104</td>
<td>6.39</td>
<td>31</td>
</tr>
</tbody>
</table>

5.3 Change-Semantic Aware Analysis

Table 3 shows the results of running our SEM-DCIA analysis on our subjects. For each change, we measure the number of lines impacted by dataflow analysis (columns DCIA Impact) and also by SEM-DCIA (columns SEM-DCIA<sub>n</sub>). The columns SEM-DCIA<sub>0</sub> denote various bounds for SEM-DCIA-ANYTIME and are discussed in § 5.4. We report for each project the minimum and maximum number of impacted lines (min, max), and for the SEM-DCIA() analysis we report also the average reduction of the size of the impacted set. Note that SEM-DCIA analysis always reports a subset of the set reported by the non-semantic analysis. We also report the average analysis time in seconds for all analyses.

Our evaluation shows that on average, the semantic-aware analysis reduces the size of the impacted set by 35%. The overhead of performing full semantic analysis on the entire program is on median 19x, ranging between 3x and 67x. While the semantic analysis results at ∞ level represent the most precise analysis our technique achieves, it is quite expensive. For example in the theft project the reduction achieved by SEM-DCIA<sub>∞</sub> is 77% but with a 64x overhead. This motivates the need for an incremental analysis, whose results are obtained faster.

**Imprecision:** Our manual inspection of results reveals three broad classes for nodes classified as impacted: (i) nodes in syntactically changed procedures, (ii) SymDiff’s inability to match loops as it relies on syntactic position in AST (this can be fixed by better matching heuristics), (iii) SMACK represents all aliased addresses accessing a field using a single map; writing to one location destroys equivalences on the map variables (need more refined conditional equivalences [26]).

5.4 Incremental Analysis

Table 3 shows the analysis results of varying the bound on k for the SEM-DCIA-ANYTIME. The first iteration SEM-DCIA<sub>0</sub> corresponds to semantically analyzing only the syntactically-changed procedures; the second iteration SEM-DCIA<sub>1</sub> corresponds to analyzing the procedures at distance at most one from the syntactically changed procedures (callers and callees). The results show that even SEM-DCIA<sub>0</sub> provides benefits, pruning the impacted set by 22% on average. The overhead is reduced compared to the full analysis (9x). The results show that the reduction in impact improves as the analysis scope (k) increases. For example, in the case of theft the improvement is from 61% (SEM-DCIA<sub>0</sub>) to 77% (SEM-DCIA<sub>∞</sub>), at the cost of overhead increase from 8x to 64x.

We find that the anytime analysis is most beneficial for cases where it is prohibitive to run the full algorithm because of time constraints. This is best illustrated for the case of space (we used a timeout of one hour). Table 4 shows the first four levels for space (two more iteration beyond the ones in Table 3); performing the analysis incrementally is still valuable even up to k = 3; the first iteration already provides big benefits on top of the non-semantic analysis, while the following iterations display a smooth improvement with each iteration. We believe this highlights the benefits of our anytime algorithm, giving the user control over the tradeoff between precision and analysis-time.

5.5 Representative Examples

Our inspection of the analysis results indicates that the improvement in precision in SEM-DCIA() comes from two fronts. First, it compensates for the price we paid for soundness by considering entire procedures as source of impact. The semantic analysis reduces the impacts for callers and callees transitively. Second, the reduction in impact happens from refactorings that a pure dataflow analysis cannot consider. We next show a few interesting patterns we discovered while applying the tool (for brevity we only describe the change briefly).

**Variable Extraction:** Figure 5 shows a refactoring to extract a constant to a variable. A non-semantic technique will create impacts in term_move_to through the first argument, since it will not be able to find that the value flowing into the first argument is the same in both versions and in all executions. Our SEM-DCIA technique will successfully prove the mutual precondition necessary to show...
Change impact analysis: Change Impact Analysis has been widely explored in static and dynamic program analysis context [10, 30, 32, 42, 45]. Most previous works perform the analysis at a coarse-grain level (classes and types) to retain soundness of analysis [1, 2, 31, 36] which can result in coarse results. JDiff [1] addresses some of the challenges of performing both a diff and computing a mapping between two programs in the context of Java object-oriented programs. Other techniques resort to dynamic information to recover from the overly-conservative dataflow analysis [2, 36]. Our goal is to improve the precision of CIA analysis by making it change-semantics aware using statically computed equivalence relations without sacrificing soundness.

Regression verification: Regression verification [20, 39] and its implementations [28] aim at proving summary equivalence inter-procedurally, but does not help with the CIA directly as shown in §1.1. The work by Bakes et al. [3] improves traditional equivalence checking by finding paths not impacted by changes through symbolic execution. The approach is non-modular (does not summarize callees), bounded (unrolls loops and recursion), and does not seek to improve the underlying change-impact analysis. The technique leverages CIA to avoid performing equivalence checking on non-impacted procedures (computed by standard dataflow analysis). These approaches are useful for equivalence-preserving changes; when the changes are non-equivalent they do not provide meaningful help for reducing code review or testing efforts. Our approach, on the other hand, refines the CIA and can be used in code review and regression testing. Besides, our approach retains modularity and is sound in the presence of loops and recursion. We leverage the product construction in SymDiff [29] that has been used for differential assertion checking (checking if an assertion fails more often after a change); however this work is limited as it requires the presence of assertions in the program. Our approach can also use other product construction techniques and relational invariant inference techniques as an off-the-shelf solver [7, 8, 11].

Symbolic Analysis: Person et al. use change-directed symbolic execution to generate regression tests [40]. Our technique can be used to prune the space for which regression tests need to be generated. In addition, there is research on relational verification using a product construction [7–9, 37], but most approaches are not automated and do not consider changes across procedure calls.

7 CONCLUSIONS

In this work, we formalize and demonstrate how to leverage equivalence relations to improve the precision of dataflow-based change-impact analysis and provide a time-precision knob, which is crucial for applying such analyses to large projects. Our work brings together program verification techniques (namely relational-invariant generation) to improve the precision of a core software engineering task, and can go a long way in providing the benefits of semantic reasoning to average developers.

ACKNOWLEDGMENTS

We thank Darko Marinov, Sasa Misailovic, August Shi, and the anonymous reviewers for their comments. Alex performed parts of this research while at Microsoft Research. He was also partially supported by the NSF Grant Nos. CCF-1421503 and CNS-1646305.
REFERENCES


Refining Interprocedural Change-Impact Analysis using Equivalence Relations

ISSTA’17, July 2017, Santa Barbara, CA, USA