Refining Interprocedural Change-Impact Analysis using Equivalence Relations

No Author

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). Existing CIA is imprecise because it is coarse-grained, deals with only few refactoring patterns, or is unaware of the change semantics.

We formalize the notion of 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 allows applying costly equivalence relation inference incrementally to refine the set of impacted statements. We have implemented a prototype in SymDiff, 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 size of the set of impacted statements compared to standard dataflow-based techniques.

1 INTRODUCTION
Software constantly evolves to add and improve features, eliminate bugs, improve software 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 [34], regression testing [19, 43], test-suite augmentation [33, 38, 39], code contracts [5, 24], regression verification [20, 37] and verification modulo versions [32]; they 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 [42] on one program to propagate the change along data and control flow edges [3, 10, 31]. 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 may require more testing.

In this paper, we aim to improve the precision of CIA by leveraging equivalence relations between 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 [45, 47] to incrementally compute equivalence relations.

1.1 Overview
Figure 1 shows a running example written 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 changes: (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 sets the line_delim variable to ‘1\0’ in the procedure print_product_info (lines 12, 13, 14), which impacts statements in print_minor_version. Assume for this example that all executions start from the procedure print_product_info. We claim that the only (syntactically unchanged) line that is impacted by the changes is the highlighted line 41 due to the semantic change at the call site; 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, the example does not contain the definitions of the setlocale and locale_format procedures as well as the LC_ALL and HEADER constants as they are not impacted nor relevant. We will now analyze the change through the lens of a standard dataflow analysis [42] and traditional equivalence checking [20, 27] and then sketch our technique.

Dataflow: A dataflow analysis technique starts at the sources of change and propagates them through data and control edges (typically in the changed program). Dataflow techniques are not aware of the change semantics, and thus cannot exploit semantics-preserving changes. Initially, the call to print_header on line 5

1Negation in C coerces the values to 0 or 1.
Figure 1: Example. The lines with − and + represent deleted and added lines, respectively.

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_major_version and print_minor_version because of the control dependency on printed and will impact all the statements in these procedures as well as all the returns at both call sites. Finally, the call to print_minor_version 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 imprecise since the analysis is unable to determine that the statements in print_name and print_major_version are not impacted.

E quivalence: A traditional interprocedural equivalence checking [20, 27] (that checks if two procedures have identical input-output behavior) will find that locale_ok, print_name, print_header, print_major_version, and print_minor_version have identical summaries. This is unsound for the question of impact analysis, as the statement of print_minor_version 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: In this work, we present a change-semantics aware CIA that works as follows: it infers equivalence relations over variables and determines that the arguments at all call sites to print_name and print_major_version are equal across 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) that arguments to print_name and print_major_version are not impacted and therefore the statements in print_name and print_major_version 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 programs (§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 [27, 28], and evaluated our technique on 322 real-world changes collected from GitHub open-source projects and several standard benchmark programs used in prior research [23].

2 BACKGROUND

For the rest of the paper, we will formalize the problem and our technique over a simple language. We can compile most features of general purpose imperative programming languages to our simple language; 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:** $e \in \text{Exprs}$ in the language are built up from constants, variables and operator applications:

$$e \in \text{Exprs} : \cdot c \mid x \mid y \ldots \mid \text{op}(e_1, \ldots, e_k)$$

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

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

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

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

**Procedures:** A procedure $f \in \text{Procs}$ is represented as a control-flow graph consisting of $(N_f, E_f, In_f, Out_f, Vars_f, n_f^0, n_f^1)$, 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) formals of $f$. The output forms model return values and out parameters.
- $Vars_f$ is the set of variables in the scope of $f$ and includes $In_f, Out_f$, and local variables of $f$.
- $n_f^0 \in N_f$ (respectively, $n_f^1 \in N_f$) is the unique entry (respectively, exit) node of $f$.

Let $N = \bigcup_{f \in \text{Procs}} N_f$ and $Vars = \bigcup_{f \in \text{Procs}} Vars_f$. 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 $RVars(n_f)$ and writset $WVars(n_f)$ as the set of variables that are read and written to respectively in the statement at $n_f$.

A program $Prog \in \text{Programs}$ is a tuple $(Procs, \text{main}, \text{StmtAt})$ where:

1. $Procs$ is a set of procedures in the program.
2. $\text{main} \in Procs$ is the entry procedure from which the program execution starts, and
3. $\text{StmtAt} : N \rightarrow \text{Stmts}$ maps a node $n \in N$ in a procedure $f$ to a statement. For any $f$, we assume that $\text{StmtAt}(n_f^0) = \text{skip}$.

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, 40].

**Control flow:** Loops can be automatically transformed into tail-recursive procedures [20, 27, 28]. We use $n_1 : st \mid \text{goto } n_2, n_3$ to express that $\text{StmtAt}(n_1) = st$ and $\{(n_1, n_2), (n_1, n_3)\} \subseteq E_f$. 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 $n \in N_f$ has at most two successor nodes in $E_f$, where nodes with two successors correspond to conditional statements. The only use of an assume statement is to model a conditional statement. We refer to $n_f$ as a branching node with two successors in $E$ 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 $\text{sel}(x,e)$ and a write $x[e] := e_2$ is modeled using $x := \text{update}(x,e_1,e_2)$ [6]. Arrays are in turn used to model the heap in imperative programs and are standard in most software verification tools [12, 18, 40]. 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 [27]. 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 $\theta\mid_{Vars_f}$ restricts the domain of the store to variables in $Vars_f$. For stores $\theta_1$ and $\theta_2$, the store $\theta_1 \oplus \theta_2$ is defined as follows for any variable $x \in \theta_1$ or $x \in \theta_2$:

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

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

**Calls:** Let $cs \in (N \times \text{Vars}^* \times \Theta)^*$ be a call stack that is a sequence of tuples $((n_0, \theta_0, \theta), (n_1, \theta_1, \theta), \ldots)$, where $n_i$ is the $i$-th call site on the call stack ($n_0$ is the most recent), $\theta_i$, 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, $\epsilon$ denotes an empty stack, and $(n, \theta, \emptyset) : cs$ 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 $\Sigma \times \Sigma$ holds only if:

1. $\text{StmtAt}(n_f) \equiv x := e, n_2 \in N_f, \theta_2 = \theta_1 \oplus [x \rightarrow \theta_1(e)], (n_f, n_2) \in E_f, \text{ and } cs_1 = cs_2$, or
2. $\text{StmtAt}(n_f) \equiv \text{assume } e, n_2 \in N_f, \theta_1(e) = \text{true}, (n_f, n_2) \in E_f, \theta_1 = \theta_2$ and $cs_2 = cs_2$, or
3. $\text{StmtAt}(n_f) \equiv \text{skip}, n_f \neq n_f', n_2 \in N_f, (n_f, n_2) \in E_f, \theta_1 = \theta_2$ and $cs_1 = cs_2$, or

The introduction of $x$ simplifies determining if control flow is impacted by only inspecting the conditional node.
Let $n_1$ be the unique successor of $n_i$ in $f$, and $\mathcal{X}$ be the vector of input forms for $g$ in $n_2 = n_i^g$, $c_{22} = (\mathcal{X}, \mathcal{X}) \triangleright c_{21}$ and $\mathcal{X}_2 = \mathcal{X} \triangleright \mathcal{X}_1$), or

$\mathcal{X}$ be the vector of output forms for $f$ in $n_2 = n_i^g$, $\mathcal{X}_2 = \mathcal{X}_1 \triangleleft \mathcal{X}_1$.

A transitive edge $e_0 \rightarrow e_1$ exists if $e_0 = e_1$ or there exists a sequence of transitions $e_0 \rightarrow e_1, \ldots, e_{n-1} \rightarrow e_n$, where $e_i \rightarrow e_{i+1}$ for all $i \in \{0, \ldots, n\}$. For procedure $f$, we denote the input-output transition relation $\Omega_f \doteq \{(e_0, e_1) \mid (n_i^f, \theta, e) \rightarrow^* (n_{i+1}^f, \theta, e_1)\}$.

**Execution Traces:** An execution trace $\tau$ is a (possibly infinite) sequence of states $(\sigma_0, \sigma_1, \ldots)$, where $\sigma_i \rightarrow \sigma_{i+1}$, for any adjacent pair of states in the sequence. For a trace $\tau$ and a node $n \in N$, $\tau\tau_{\pi_n}$ denotes the (maximal) subsequence of $\tau$ containing states of the form $(n_{\pi_n}, \cdot)$. For such a trace $\tau$ of length at least $i + 1$, $\tau[i]$ denotes the state at position $i$ (namely $\sigma_i$). For any procedure $f$, let $\Gamma_f$ be the set of maximal traces of $f$. That is, $\Gamma_f$ is the set of all traces $\tau$ such that (i) $\tau[0] = (n_i^f, e)$, and (ii) either (a) the final state $\sigma_n$ has no successors, or (b) the trace is non-terminating. Traces with no successors can either terminate normally in a state $(n_{\pi_n}, e)$, or could be blocked due to no successors in $E$ or due to an unsatisfied assume statement. For a store $\theta \in \Theta$, we denote $\tau(\theta)$ as the maximal trace (due to determinism) of $f$ that starts in a store $\theta$.

### 3 PROBLEM STATEMENT

In this section, we formalize the problem of semantic change-impact analysis and provide a solution based on dataflow-based static analysis.

#### 3.1 Representing Changes

We denote $Prog^1, Prog^2 \in Programs$ as two versions of a program. Similarly $\sigma^1, \theta^2, \tau^1, Procs^1, main^1, StmtAt^1$ denote entities for $Prog^1$, without making $Prog^2$ explicit.

To simplify the formulation we assume the two programs in a normalized form, where (i) each procedure in $Procs^2$ has a corresponding procedure in $Procs^1$ and vice versa, and (ii) for each $f \in Procs^2$, the vector of variables in $Vars_f$ and the set of nodes $N_f$ (but not necessarily $E_f$) are identical with the ones in the corresponding procedure. We can easily preprocess the programs to obtain their normalized form, by introducing additional procedures, variables (initialized and) and nodes. Finally, for any missing node $n$, we add an unreachable node in $N_f$ with a skip statement and empty successor list.

**Difffing:** Given the two versions, a diffing algorithm produces a mapping between nodes in the two programs. We assume we are given a sound diff algorithm to label the sources of change. A diff algorithm is sound if it produces a partial function $\pi: N_1 \rightarrow N_2$ such that:

1. $\pi$ is a partial bijection and $StmtAt(n_i) = StmtAt(\pi(n_i))$.
2. For any two traces $\tau^1 = \tau^1_{main}(\theta)$ in $Prog^1$ and $\tau^2 = \tau^2_{main}(\theta)$ in $Prog^2$, $\tau^1$ only executes statements in $Dom(\pi)$ iff $\tau^2$ only executes statements in $Im(\pi)$.

A partial bijection is a partial function that is injective when defined and (trivially) surjective when restricted to its image [21].

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

<table>
<thead>
<tr>
<th>Predicate name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchingNode(n)</td>
<td>if $n$ is a branching node</td>
</tr>
<tr>
<td>controlDependent(n$_2$, n$_1$)</td>
<td>if $n_2$ is control-dependent on $n_1$ [17]</td>
</tr>
<tr>
<td>callee(n, f, g)</td>
<td>if StmtAt(n) is a call to $f$ within a caller $g$.</td>
</tr>
<tr>
<td>inFormal(x, i, f)</td>
<td>if $x$ is the $i$-th input formal of $f$</td>
</tr>
<tr>
<td>outFormal(x, i, f)</td>
<td>if $x$ is the $i$-th output formal of $f$</td>
</tr>
<tr>
<td>inActual(e, i, f, n)</td>
<td>if the expression $e$ is the $i$-th actual argument to a call to $f$ at a callsite $n$</td>
</tr>
<tr>
<td>outActual(r, i, f, n)</td>
<td>if the variable $r$ receives the $i$-th output formal to a call to $f$ at a callsite $n$</td>
</tr>
</tbody>
</table>

(3) For any two traces $\tau^1 \doteq \tau^1_{main} (\theta)$ in $Prog^1$ and $\tau^2 \doteq \tau^2_{main}(\theta)$ in $Prog^2$, where $\tau^1$ only executes statements in $Dom(\pi)$ or $\tau^2$ only executes statements in $Im(\pi)$, then $\tau^1 \doteq \tau^2$.

The mapped nodes $\mathit{mapped} \doteq \mathit{Dom}(\pi) \cup \mathit{Im}(\pi)$ underapproximate the set of nodes that are syntactically unchanged. Intuitively, if a program executes only statements in $\mathit{mapped}$ then the program behaves the same in both versions; statements that are not in $\mathit{mapped}$ are the sources of change.

We describe for illustrative purposes a simple diffing algorithm which is sound. The algorithm proceeds to produce a mapping $\pi$ as follows: Let $Procs^A \subseteq Procs$ be the set of procedures that have some syntactic change. Any node not in $f \in Procs^A$ trivially mapped as the control-flow graphs are identical in the two versions. Any node in $f \in Procs^A$ is conservatively treated as not mapped. Our formulation is parameterized by a diff algorithm which can be either based on text [46] or more sophisticated notions such as abstract syntax trees [16] or program-dependency-graphs [26] as long as they satisfy the soundness criteria.

#### 3.2 Semantic Change Impact

We can now state the meaning of a node being impacted by a program change, in terms of the trace semantics of the two programs and the set $\mathit{mapped}$.

For a sequence of states $\bar{\sigma}$ and a variable $x \in Vars$, $\bar{\sigma} x \in (\sigma \cup (\perp))$ denotes the sequence of values $\bar{\sigma}$ with the same length as $\bar{\sigma}$, and $v_i = \begin{cases} \theta(x), & \sigma_i \doteq (\perp, \perp, \perp) \\ \perp, & \text{otherwise} \end{cases}$

**Definition 3.1 (Impacted nodes).** Given $Prog^1, Prog^2$ and $\mathit{mapped}$, a node $n \in N^1 \cup N^2$ is impacted if either $\mathit{Impacted}(n, Prog^1, Prog^2, \pi)$ or $\mathit{Impacted}(\pi(n), Prog^2, Prog^1, \pi^{-1})$ holds, where $\pi^{-1}$ is the inverse. $N^1$ is the corresponding $N$ for $Prog^1$.

We define $\mathit{Impacted}(k, Prog^a, Prog^b, \pi)$:

1. $k \notin \mathit{Dom}(\pi)$, or
2. there exists a store $\theta$, pair of traces $\tau^a \doteq \tau^a_{main}(\theta)$ for $Prog^a$ and $\tau^b \doteq \tau^b_{main}(\theta)$ for $Prog^b$, and a variable $x \in RVars(n)$ such that $(\tau^a_k)_{\theta} \neq (\tau^b_k)_{\theta(k)}$.
We conservatively treat any unmapped node as impacted. A mapped node \( n \) is not impacted if the sequence of values of variables in \( RVars(n) \) is identical for any two execution traces \( \tau^a \) (in \( Prog^a \)) and \( \tau^b \) (in \( Prog^b \)) starting from a common input store \( \theta \) to \( main \). Note that for our low-level language, the \( RVars(n) \) of a statement often includes the state of the heap and address being written to. For example, the C# statement \( x.length = y \) gets translated to \( n : Length := update(Length,x,y) \), (where \( Length \) is an array variable representing the state of length field/attribute in all objects) with \( RVars(n) = \{ Length, x,y \} \).

### 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 [42] that starts with a program \( Prog \) \((i \in \{1,2\}) \) 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 \( \text{OutActual}(r,i,f,n) \) 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 \( \text{CONTROLDEPEND}(n_1,n_2) \), a node \( n_2 \) is control-dependent on node \( n_1 \) if (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].

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{DEPENDS-ENTRY}</td>
<td>( x \in \text{In}_{f_i} )</td>
</tr>
<tr>
<td>\text{DEPENDS-WRITE}</td>
<td>( y \in WVars(n) ) ( \in \text{W}_{f_i} )</td>
</tr>
<tr>
<td>\text{DEPENDS-TRANSITIVE}</td>
<td>( \text{DEPENDSOnVar}(y,x,f) )</td>
</tr>
<tr>
<td>\text{CONTROL-DEPENDS}</td>
<td>( \text{CONTROLDEPEND}(n_2,n_1) )</td>
</tr>
<tr>
<td>\text{SUMMARY-DEPENDS}</td>
<td>( \text{OutActual}(r,i,f,n) )</td>
</tr>
<tr>
<td>\text{DEPENDS-OUTPUT}</td>
<td>( \text{DEPENDSOnVar}(y,x,f) )</td>
</tr>
</tbody>
</table>

**Figure 2**: Inference rules for computing \( \text{DEPENDSOnVar} \) and \( \text{DEPENDSOnNode} \). The input is a program \( Prog \).

**Dependency**: Figure 2 describes a set of inference rules to compute two relations \( \text{DEPENDSOnVar} \) and \( \text{DEPENDSOnNode} \). For a pair of variables \( x,y \in \text{Vars}_f \) such that \( y \) is either data- or control-dependent on \( x \), then \( \text{DEPENDSOnVar}(y,x,f) \) holds. Similarly, a node \( n \in \text{N}_{f_j} \) and a variable \( y \) such that \( y \) is data or control dependent on a variable \( x \) that is updated at \( n \), \( \text{DEPENDSOnNode}(y,n,f) \) holds. An inference rule (e.g. \( \text{DEPENDS-NODE} \)) 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 \( \text{CONTROL-DEPENDS} \) 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 \( \text{SUMMARY-DEPENDS} \) captures the dependency of an actual return \( r \) on a variable \( w \) passed as an argument to \( f \) in a \( \text{call} \) \( g \), where \( w \) indirectly flows to \( r \) through a procedure \( \text{call} f \). For this callsite, 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 \( a \) at the callsite.

**Impact Analysis**: Figure 3 describes a set of inference rules to compute the set of nodes that are impacted in either program. For this section, we will ignore the highlighted antecedents (they become relevant in § 4 where we describe how we incorporate change semantics). The rules take as input a program (either \( Prog^3 \) or \( Prog^4 \)), 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 represent an upper bound on the set of impacted nodes, impacted variables and impacted variable summaries, respectively. Most rules are self-explanatory; we describe the main rules relevant to the interprocedural reasoning. Note that we do not have a special rule for control-flow impact, since \( \text{DEPENDSOnVar} \) already captures control flow dependency.

For an output formal \( y \in \text{Out}_{f_i} \) the summary (input-output dependency) may change either when (i) \( y \) depends on a variable updated at an unmapped node \( n \in \text{N}_{f_j} \) (expressed by \( \text{IMPACT-SUMMARY} \)), or (ii) \( y \) depends on a variable \( w \) that stores the output formal \( x \) of \( g \) at a callsite in node \( n \), and the summary of \( x \) has changed in \( g \) (expressed by \( \text{IMPACT-SUMMARY-PROP} \)).

The CALL-IMPACT rule says that an input formal \( x \) in \( f \) can be impacted if the corresponding actual argument \( e \) at a callsite is impacted. RETURN-IMPACT considers the case when the variable summary for the corresponding output formal \( y \) is impacted. SUMMARY-IMPACT considers the case when the actual argument expression \( e \) (passed for the formal \( x \) of \( f \)) is impacted, and \( y \) depends on the value of \( x \) in \( f \). 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 \( Prog^3 \), \( Prog^4 \) and \( MAPPED \).
(2) Applies the inference rules in Figure 3 on \( Prog^4 \) to generate \( \text{IMPACTEDNODE}^4 \), \( \text{IMPACTEDVAR}^4 \), \( \text{IMPACTEDSUMM}^4 \) until a fix-point is reached.
(3) Returns the tuple \( (\bigcup_i \text{IMPACTEDNODE}^i, \bigcup_i \text{IMPACTEDVAR}^i, \bigcup_i \text{IMPACTEDSUMM}^i) \).
The following theorem states the soundness of the dataflow analysis DCIA.

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

Consider 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_version, and print_minor_version. 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 DependsOnVar).

**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 Prog$^1$ and Prog$^2$.

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

**Definition 4.1 (PreEquiv).** PreEquiv($x, f$) holds for an input formal $x \in$ In$_{f}$ if for all stores $\theta$, and for every pair of traces $r^1 = r^1_{\text{main}}(\theta)$ and $r^2 = r^2_{\text{main}}(\theta)$, $(r^1 | r^2_{\text{main}}) |x = (r^2 | r^2_{\text{main}}) |x$.

Intuitively, PreEquiv($x, f$) holds for an input formal $x$ of $f$ if any two executions starting from the same input $\theta$ to main call $f$ with the same sequence of values of $x$. For example, in Figure 1 the equivalences PreEquiv(delim, print_header), PreEquiv(locale, print_name), PreEquiv(locale, print_major_version), and PreEquiv(locale, print_minor_version) hold. In contrast, PreEquiv(delim, print_minor_version) does not hold, because of different values for delim ‘\n’ and ‘\n’ respectively, at the call-site in print_product_info.

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

**Definition 4.2 (SummaryEquiv).** SummaryEquiv($y, f$) holds for an output formal $y \in$ Out$_{f}$ if $(\theta_1, \theta_2) \in$ Omega in Prog$^1$ and $\theta_1 =_{\text{vars}} \theta_2$, then $(\theta_2, \theta_4) \in$ Omega in Prog$^2$ ($j \neq i$) and $\theta_4(y) = \theta_4(y)$.

Intuitively, if the versions of $f$ are executed from stores $\theta_1$ and $\theta_2$ where $\theta_1 =_{\text{vars}} \theta_2$, 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 print_product_info, i.e., in this case SummaryEquiv(line_delim, print_product_info) does not hold since in one version the value of line_delim at the end of the execution is “\n” while in the other it is undefined.
Applying DCIA change does not propagate outside the changed statement. For example, the strengthened CALL-IMPACT prevents an input formal \(x\) from being impacted if \(\text{PREEQUIV}(x, f)\) holds. Similarly, the strengthened IMPACT-SUMMARY prevents a summary for \(y\) from impact if we know that \(\text{SUMMARYEQUIV}(y, f)\) holds. The strengthened 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 \(\text{Prog}^1, \text{Prog}^2 \in \text{Programs}, \text{MAPPED}, \text{PREEQUIV}, \text{and SUMMARYEQUIV}, (i) SEM–DCIA terminates, and (ii) for any \(n \notin \text{IMPACTED NODE}, n\) is not an impacted node with respect to 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 [28]. 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.

To exploit the change semantics, it is often useful to apply equivalence relation inference only in the vicinity of actual syntactic changes.

```c
void main(int x) {
    \(f_1(x)\);
    \(f_1(x+2)\);
}

void \(f_1\) (int x) {
    \(f_1(x+1)\);
}
```

**Figure 4:** Motivating example for anytime algorithm.

Consider the example in Figure 4 to make the intuition clear. Applying DCIA will result in impacting all the nodes in the program as follows. The modified call node for \(f_1\) in \textit{main} is not in \textit{MAPPED}, which impacts input formal \(x\) of \(f_1\). This in turn impacts the call to \(f_2\) and so on. On the other hand, we can observe that \(\text{PREEQUIV}\) and \(\text{SUMMARYEQUIV}\) hold for each of the procedures because the change does not propagate outside the changed statement.

For Figure 4 it suffices to infer the equivalences on \textit{main} while abstracting the rest of the procedures from the expensive equivalence analysis. Considering \(f_1\) has all callsite inside \textit{main} and that it does not have an impacted summary by rule IMPACT-SUMMARY after DCIA suffices to determine that \(\text{PREEQUIV}(x, f_1)\) holds. This information can be fed to SEM–DCIA which will prune the impact for the input parameter of \(f_1\) which will prune the remaining impacts when performing a pure dataflow analysis. Thus, we obtain a precise change-impact analysis by applying the equivalence inference only on a small subset of the procedures in the program. Similarly, in Figure 1 it suffices to analyze only the syntactically changed procedures and abstract away the others to obtain the most precise result; this is not the case in general because to infer the \(\text{PREEQUIV}\) we need all call sites to be in scope, not only the syntactically changed procedures.

**Algorithm 1:** SEM–DCIA-ANYTIME

```c
\begin{verbatim}
begin
\(k \leftarrow 0;\)
\(\text{EQ} \leftarrow (0,0);\)
\((\text{impNds}, \text{impVars}, \text{impSumms}) \leftarrow \text{SEM–DCIA}(\text{Prog}^1, \text{Prog}^2, \text{MAPPED}, \text{EQ});\)
\text{Procs}' \leftarrow \text{Procs};
while \(\text{Procs}' \subseteq \text{Procs}\) do
    \(\text{prEQ} \leftarrow \{(x,f) \mid x \in \text{Inf} \text{ and } x \notin \text{impVars}\};\)
    \(\text{smEQ} \leftarrow \{(x,f) \mid x \in \text{Outf} \text{ and } x \notin \text{impSumms}\};\)
    \(\text{EQ} \leftarrow \text{EQ} + (\text{prEQ}, \text{smEQ});\)
    \(\text{Procs}' \leftarrow \text{ProcsWithin}(\text{Procs}', \text{Prog}^1, \text{Prog}^2, k);\)
    \(\text{Procs}' \leftarrow \text{AbstractProcs}(\text{Procs}', \text{Prog}^1, \text{Procs}' \setminus \text{Procs}');\)
    \(\text{EQ} \leftarrow \text{InferEquivs}(\text{Procs}', \text{Prog}^2, \text{EQ});\)
    \((\text{impNds}, \text{impVars}, \text{impSumms}) \leftarrow \text{SEM–DCIA}(\text{Prog}^1, \text{Prog}^2, \text{MAPPED}, \text{EQ});\)
    \(k++;\)
return \text{impNds};
end
\end{verbatim}
```

Algorithm 1 (SEM–DCIA-ANYTIME) provides an anytime algorithm that performs the integration. The algorithm takes as an additional input \(\text{Procs}'\), 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, 45, 47] 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 \(\text{EQ}\) (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 \text{Procs}' around \text{Procs}' that are analyzed for inferring equivalences in \text{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 \( Procs^3 \) within a call stack of depth \( k \); \( k \) is incremented with each iteration of the loop. \( AbstractProcs \) abstracts all procedures outside \( Procs' \); it only retains the knowledge of whether any procedure \( f \in Procs' \) has additional call sites outside \( Procs' \) - this determines whether \( PreEquiv \) can be inferred for a procedure. \( InferEquivs \) is invoked with a set of equivalences in \( EQ \) on the smaller programs \( Prog_4' \). 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 \( Procs' \) 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 \( SEM-DCIA_k \) as an instantiation of the algorithm \( SEM-DCIA-ANYTIME \) that terminated after the loop is executed exactly \( k + 1 \) times. We also denote \( SEM-DCIA_{\infty} \) if the loop terminates normally after \( Procs' \) equals \( Procs \).

**Theorem 4.4 (Soundness).** Given \( Prog^3, Prog^2 \in Programs, MAPPED, \) and \( Procs^3 \), if \( SEM-DCIA_k \) terminates then for any \( n \notin impNds \), \( n \) is not an impacted node with respect to \( MAPPED \) (according to Definition 3.1).

### 5 IMPLEMENTATION AND EVALUATION

#### 5.1 Implementation

We implemented our \( SEM-DCIA \) analysis for \( C \) programs, but our analysis is implemented over the intermediate verification language Boogie [4]. We leverage SMACK [40] to convert LLVM bytecode to Boogie programs.

**Differing:** For our initial implementation, we leveraged \( Diff \) over \( C \) files to produce the source 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 \( Prog^4 \) and \( Prog^3 \), SymDiff generates a product program \( Prog^{1\times2} \) that defines a procedure \( \beta^{1\times2} \) for every \( f \) and \( \pi(f) \in Procs^3 \). For the product program \( Prog^{1\times2} \), one can leverage any of the (single program) invariant generation techniques to infer preconditions, postconditions (including two-state postconditions) on \( \beta^{1\times2} \). Such invariants are relational in that they are over the state of two programs \( Prog^4 \) and \( Prog^3 \), and include equivalences relations such as \( PreEquiv \) (preconditions of \( \beta^{1\times2} \)) and \( SummaryEquiv \) (summary of \( \beta^{1\times2} \)). To ensure our inferred equivalences are valid we require the programs to be equi-terminating [22]; 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 equalities from DCIA.

#### 5.2 Evaluation

In this section we demonstrate the effectiveness of our approach on GitHub projects with real program changes and also 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. The size of the impacted set is a proxy metric for the effort necessary to perform many software engineering tasks such as code review and testing.

We analyze 164 actual changes consisting of refactorings, feature additions, buggy changes, and bug fixes from 5 projects from the GitHub repository. We selected the projects based on popularity, size, active development, and based on compatibility with SMACK. The projects, number of versions used, their size in source lines of code (SLOC), and corresponding change sizes (in number of \( C \) source lines changed) are summarized in Table 2. Our subjects are applications written in \( C \) such as 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). In addition, we also include 6 standard benchmark programs widely used by prior research on regression testing [23]. 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 on the small end of this spectrum).

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).

#### 5.3 Change-Semantic Aware Analysis

Table 3 shows the results of running our \( SEM-DCIA \) analysis on each of our subjects. For each change, we measure the number of lines reported as impacted by dataflow analysis (columns DCIA

<table>
<thead>
<tr>
<th>Project Name</th>
<th># Version Pairs</th>
<th>SLOC Changed min</th>
<th>SLOC Changed 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>1838</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>299</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 2: Summary of projects used as evaluation subjects
Impact and also by SEM-DCIA (columns SEM-DCIA∞). The columns SEM-DCIA denote various bounds for SEM-DCIA-ANYTIME and results will be 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 the non-semantic analysis and for the SEM-DCIA analysis.

Our evaluation shows that on average, the change-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 \( \infty \) level represent the most precise analysis our technique achieves, it is quite expensive, and it even times-out for our largest program (e.g. space). For example in the theft project the reduction achieved by SEM-DCIA∞ is 77% but with a 64x overhead. This motivates the need for an incremental analysis, whose results can be 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 [25]).

### 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-DCIA0 corresponds to semantically analyzing only the syntactically-changed procedures; the second iteration SEM-DCIA1 corresponds to analyzing the procedures at distance at most one from the syntactically changed procedures (callers and callees). The results show that even SEM-DCIA0 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-DCIA0) to 77% (SEM-DCIA∞), 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 upto \( k = 3 \); 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(\( k \)) 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...

<table>
<thead>
<tr>
<th>Project Name</th>
<th>DCIA min</th>
<th>DCIA max</th>
<th>Time</th>
<th>SEM-DCIA0 min</th>
<th>SEM-DCIA0 max</th>
<th>Red</th>
<th>Time</th>
<th>SEM-DCIA1 min</th>
<th>SEM-DCIA1 max</th>
<th>Red</th>
<th>Time</th>
<th>SEM-DCIA∞ min</th>
<th>SEM-DCIA∞ max</th>
<th>Red</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>flingd</td>
<td>64</td>
<td>84</td>
<td>0.94</td>
<td>39</td>
<td>83</td>
<td>20.1%</td>
<td>8.92</td>
<td>14</td>
<td>70</td>
<td>47.3%</td>
<td>8.95</td>
<td>14</td>
<td>70</td>
<td>47.3%</td>
<td>10.44</td>
</tr>
<tr>
<td>histo</td>
<td>86</td>
<td>2.14</td>
<td>0</td>
<td>75</td>
<td>11.5%</td>
<td>19.43</td>
<td>0</td>
<td>63</td>
<td>28.6%</td>
<td>20.59</td>
<td>0</td>
<td>65</td>
<td>28.6%</td>
<td>24.92</td>
<td></td>
</tr>
<tr>
<td>mdp</td>
<td>0</td>
<td>465</td>
<td>28.16</td>
<td>0</td>
<td>330</td>
<td>1.5%</td>
<td>77.71</td>
<td>0</td>
<td>324</td>
<td>3.4%</td>
<td>100.68</td>
<td>0</td>
<td>283</td>
<td>6.5%</td>
<td>173.08</td>
</tr>
<tr>
<td>tinyvm</td>
<td>0</td>
<td>344</td>
<td>68.96</td>
<td>0</td>
<td>308</td>
<td>18.3%</td>
<td>158.35</td>
<td>0</td>
<td>298</td>
<td>23.2%</td>
<td>160.35</td>
<td>0</td>
<td>283</td>
<td>43.6%</td>
<td>169.05</td>
</tr>
<tr>
<td>theft</td>
<td>184</td>
<td>261</td>
<td>4.48</td>
<td>11</td>
<td>186</td>
<td>61%</td>
<td>38.45</td>
<td>11</td>
<td>185</td>
<td>62%</td>
<td>57.48</td>
<td>11</td>
<td>107</td>
<td>77%</td>
<td>208.75</td>
</tr>
<tr>
<td>print_tokens</td>
<td>151</td>
<td>153</td>
<td>2.22</td>
<td>69</td>
<td>137</td>
<td>19.57%</td>
<td>24.02</td>
<td>34</td>
<td>128</td>
<td>28.67%</td>
<td>58.23</td>
<td>34</td>
<td>128</td>
<td>28.67%</td>
<td>102.40</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>155</td>
<td>158</td>
<td>1.46</td>
<td>80</td>
<td>129</td>
<td>30.36%</td>
<td>16.08</td>
<td>59</td>
<td>101</td>
<td>44.65%</td>
<td>24.42</td>
<td>55</td>
<td>100</td>
<td>45.66%</td>
<td>97.98</td>
</tr>
<tr>
<td>replace</td>
<td>75</td>
<td>195</td>
<td>4.96</td>
<td>74</td>
<td>194</td>
<td>2.08%</td>
<td>35.72</td>
<td>70</td>
<td>194</td>
<td>2.89%</td>
<td>92.72</td>
<td>65</td>
<td>174</td>
<td>9.41%</td>
<td>236.77</td>
</tr>
<tr>
<td>schedule</td>
<td>79</td>
<td>115</td>
<td>1.37</td>
<td>7</td>
<td>104</td>
<td>26.35%</td>
<td>13.75</td>
<td>7</td>
<td>87</td>
<td>40.58%</td>
<td>24.15</td>
<td>7</td>
<td>68</td>
<td>79.85%</td>
<td>30.83</td>
</tr>
<tr>
<td>space</td>
<td>20</td>
<td>2851</td>
<td>59.45</td>
<td>14</td>
<td>2816</td>
<td>31.87%</td>
<td>798.96</td>
<td>14</td>
<td>2816</td>
<td>36.71%</td>
<td>895.14</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>timeout</td>
</tr>
<tr>
<td>tcas</td>
<td>1</td>
<td>49</td>
<td>0.66</td>
<td>0</td>
<td>49</td>
<td>9.24%</td>
<td>7.94</td>
<td>0</td>
<td>49</td>
<td>9.24%</td>
<td>8.63</td>
<td>0</td>
<td>49</td>
<td>9.24%</td>
<td>9.71</td>
</tr>
<tr>
<td>tot_info</td>
<td>103</td>
<td>104</td>
<td>6.39</td>
<td>31</td>
<td>102</td>
<td>18.65%</td>
<td>37.01</td>
<td>24</td>
<td>102</td>
<td>46.26%</td>
<td>74.99</td>
<td>12</td>
<td>77</td>
<td>56.50%</td>
<td>127.61</td>
</tr>
</tbody>
</table>

**Table 3:** Analysis results for different levels of precision. Time in seconds. (timeout = 1 hour)

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Min</th>
<th>Max</th>
<th>Reduction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCIA</td>
<td>20</td>
<td>2851</td>
<td>n.a.</td>
<td>59.45</td>
</tr>
<tr>
<td>SEM-DCIA0</td>
<td>14</td>
<td>2816</td>
<td>31.87%</td>
<td>798.96</td>
</tr>
<tr>
<td>SEM-DCIA1</td>
<td>14</td>
<td>2816</td>
<td>36.71%</td>
<td>895.14</td>
</tr>
<tr>
<td>SEM-DCIA2</td>
<td>14</td>
<td>2816</td>
<td>40.56%</td>
<td>1300.43</td>
</tr>
<tr>
<td>SEM-DCIA3</td>
<td>14</td>
<td>2808</td>
<td>43.96%</td>
<td>1900.03</td>
</tr>
<tr>
<td>SEM-DCIA∞</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

**Table 4:** Analysis results for space
```c
void draw_histogram(int data[], int len) {
    int xbarw = 5;
    while (y--) {
        term_move_to(x * 5 + xpad + 3,
                    term_move_to(x * xbarw + xpad + 3,
                                  y - 1 + h + ypad);
        }
    }
}
```

**Figure 5:** Change illustrating an extract constant to variable in histo commit e723a4

```c
-while (*c) {
+for (;*c;c++) {
    wprintw(window, "%c", *c);
    c++;
}
```

**Figure 6:** Change illustrating a loop conversion in mdp commit 00c2ad

```c
- if (!strend || !strbegin) goto pp_ret;
+ if (!strend || !strbegin) return 0;
   if (!pFile) {
      ...
   goto pp_ret;
+   return 0;
   }
```

**Figure 7:** Change illustrating a goto-elimination refactoring in tinyvm commit 378cc6

the equality in both versions, and hence cut impacts that would propagate through the first argument.

**Loop Refactoring:** Figure 6 shows a change from a while loop to a for loop. Remember that we extract loops as tail recursive procedures. Input-output equivalence checking would not prevent the impact of the argument c to the callee inside the loop–the body of the loop–(nor would dataflow analysis).

**Control-Flow Equivalence:** Figure 7 shows a change to replace a goto with return statements. This is a change in the project tinyvm. The goto statements were all redirecting control-flow to a return statement, so the developer replaced the goto with the target return statement. Our semantic technique successfully finds that the change does not produce impacts.

### 6 RELATED WORK

Our work is closely related to work aiming to support developers in evolution tasks through change-impact analysis, regression verification, regression test generation.

#### Change impact analysis:
Change Impact Analysis has been widely explored in static and dynamic program analysis context [10, 29, 31, 41, 44]. Most previous works perform the analysis at a coarse-grain level (classes and types) to retain soundness of analysis [1, 2, 30, 35] 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, 35]. Our goal is to improve the precision of CIA analysis by making it change-semantic aware using statically computed equivalence relations without sacrificing soundness.

**Regression verification:** Regression verification [20, 38] and its implementations [27] 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 [28] 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].

**Regression testing:** Person et al. using change directed symbolic execution to generate regression tests [39]. Our technique can be used to prune the space of statements for which regression tests need to be generated. In addition, there is research on relational verification using a product construction [7–9, 36], but most approaches are not automated and do not consider changes across procedure calls.

### 7 CONCLUSIONS

In this work, we have formalized and demonstrated how to leverage equivalence relations to improve the precision of change-impact analysis and provide a scalability-precision knob with SEM-DCIA-ANYTIME, 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 deep semantic reasoning to average developers.
REFERENCES


