A Change Impact Analysis to Characterize Evolving Program Behaviors

Neha Rungta
NASA Ames Research Center
Email: neha.s.rungta@nasa.gov

Suzette Person
NASA Langley Research Center
Email: suzette.person@nasa.gov

Joshua Branchaud
University of Nebraska-Lincoln
Email: jbrancha@cse.unl.edu

***Abstract*—Change impact analysis techniques estimate the potential effects of changes made to software. Directed Incremental Symbolic Execution (DiSE) is an intraprocedural technique for characterizing the impact of software changes on program behaviors. DiSE first estimates the impact of the changes on the source code using program slicing techniques, and then uses the impact sets to guide symbolic execution to generate path conditions that characterize impacted program behaviors. DiSE, however, cannot reason about the flow of impact between methods and will fail to generate path conditions for certain impacted program behaviors. In this work, we present iDiSE, an extension to DiSE that performs an interprocedural analysis. iDiSE combines static and dynamic calling context information to efficiently generate impacted program behaviors across calling contexts. Information about impacted program behaviors is useful for testing, verification, and debugging of evolving programs. We present a case-study of our implementation of the iDiSE algorithm to demonstrate its efficiency at computing impacted program behaviors.**

Traditional notions of coverage are insufficient for characterizing the testing efforts used to validate evolving program behaviors because they do not take into account the impact of changes to the code. In this work we present novel definitions of impacted coverage metrics that are useful for evaluating the testing effort required to test evolving programs. We then describe how the notions of impacted coverage can be used to configure techniques such as DiSE and iDiSE in order to support regression testing related tasks. We also discuss how DiSE and iDiSE can be configured for debugging; finding the root cause of errors introduced by changes made to the code. In our empirical evaluation we demonstrate that the configurations of DiSE and iDiSE can be used to support various software maintenance tasks.

I. Introduction

Change impact analysis techniques [1] estimate the potential effects of software changes. The results of an impact analysis can be used by other program analysis techniques, e.g., regression testing, to determine which parts of a program to re-analyze and which parts can safely be ignored because they are not impacted by the changes. The evolutionary nature of software development, coupled with the size and complexity of deployed software systems motivates the need for efficient change impact analyses. It is a well known fact that a "one line fix" can have broad and unintended (and potentially disastrous) consequences, thus impact analysis techniques play an important role in software evolution and maintenance activities.

978-1-4673-2312-3/12/$31.00

Most existing automated impact analysis techniques characterize the effects of changes in terms of syntactic program structures, such as functions or program statements, that may be impacted. Dependency-based impact analysis techniques [2] estimate the effects of program changes by analyzing the interconnections between program components. Given a *change set*—the set of program components that are known to be changed—these techniques compute the *impact set*—the set of program components that may be impacted by the actual changes. This type of characterization describes the impact of changes in terms of program locations, but does not include descriptions of feasible program execution paths through the impacted locations. Such details are useful for verification, validation and debugging of evolving program behaviors to reduce the scope of the analysis to focus on only the set of impacted program behaviors.

Directed Incremental Symbolic Execution (DiSE) is an intraprocedural change impact analysis for characterizing the impact of software changes on program execution behaviors [7]. DiSE combines the efficiencies of static analysis with the precision of symbolic execution [8], [9]. The static analysis component of DiSE uses the locations of the actual changes with program slicing techniques to estimate the impact of the changes on other locations in the source code. The resulting impact set is then used to guide symbolic execution to explore program execution behaviors impacted by the changes, and to generate path conditions summarizing the impacted program behaviors. Characterizing impacted program behaviors in terms of path conditions has the advantage that path conditions can be checked using Satisfiability Modulo Theories (SMT) solvers in order to support various software maintenance tasks such as testing, debugging, and verification.

DiSE is an intraprocedural analysis and does not account for the flow of impact between methods. Most realistic programs, however, consist of multiple methods that are invoked from different calling contexts; information flows between methods through method arguments and return values. In this work we present an extension to DiSE that performs an interprocedural analysis enabling analysis of whole programs. This extension of DiSE – iDiSE – combines static and dynamic calling context information to efficiently generate impacted program behaviors across calling contexts. Supporting the flow of impacted behaviors across methods in interprocedural programs is a non-trivial part of our analysis. We demonstrate

the efficiency of our change impact analysis with an evaluation of our implementation of iDiSE.

Many change impact analysis techniques are developed with a particular software evolution task in mind, e.g., regression testing [3]. Some techniques are configurable in that they provide options to adjust the precision of the analysis, e.g., [4]. In this work, we illustrate how DiSE and iDiSE results can be used by multiple client analyses, e.g., testing, debugging. First, we describe novel notions of coverage that incorporate change impact information. These notions of coverage can enable us to evaluate the test effort required to validate evolving program behaviors. Traditional coverage notions, e.g., statement and branch coverage, are extended to take into account change impact information. The notions of impacted coverage can then be used to configure techniques such as DiSE and DiSE in order to support regression testing related tasks. We also describe how DiSE can be configured for debugging—finding the root cause of errors introduced by changes to the code. The primary contributions of our work are:

- An extension to the DiSE algorithm to support interprocedural change impact analysis (iDiSE).
- Extended notions of coverage that combine traditional notions of coverage with change impact information.
- Configuration options for DiSE and iDiSE based on the requirements of the client analysis.
- An implementation of iDiSE as an extension to the Java PathFinder symbolic execution framework [10], [11], [12].
- An empirical evaluation illustrating the effectiveness of (a) iDiSE at characterizing the impact of change on program execution behaviors, and (b) various DiSE and iDiSE configurations to support regression testing tasks based on impacted coverage notions and debugging.

II. Background

In this section, we provide an overview of the DiSE methodology [7] and illustrate, using a small example, how the intraprocedural analysis estimates which behaviors may be impacted by changes to a method. We begin with a brief explanation of symbolic execution.

A. Symbolic Execution

Symbolic execution is a non-standard approach for executing programs that uses symbolic values in place of concrete (actual) values as program inputs [8], [9]. During symbolic execution, each program statement is executed and the symbolic state is updated to represent the effects on program variables. At every conditional branch in the program, a constraint is generated and added to the current *path condition*. A path condition is a conjunction of constraints over constants and symbolic input values that characterizes the current execution path. The satisfiability of the path condition is checked each time it is updated in order to determine the feasibility of the current program path. When a path becomes infeasible, symbolic execution stops exploration of that path and backtracks

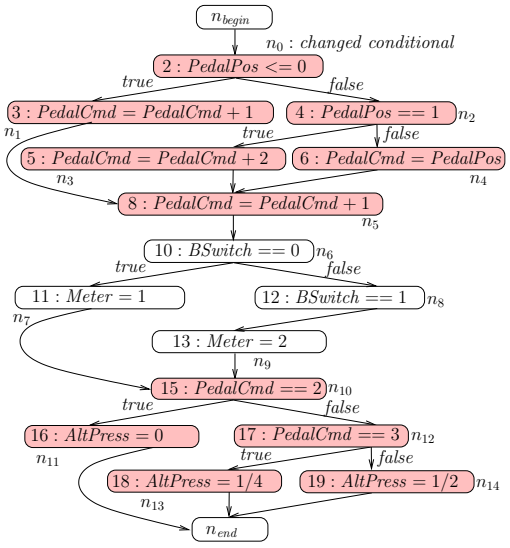

Fig. 1. Control flow graph for a simplified version of a Wheel Brake System. Node n_0 in the graph is a changed program location while the shaded nodes are impacted program locations.

to explore another path. The resulting set of path conditions form a summary of the program execution behaviors.

B. Motivating Example

To illustrate the DiSE analysis, a control flow graph (CFG) for a simplified method in a Wheel Brake System (WBS) system is shown in Fig. 1. The WBS example is taken from [7]; it contains two integer global variables, `AltPress` and `Meter`. The CFG shown is for the `update` method in WBS with three integer variables, `PedalPos`, `BSwitch`, and `PedalCmd`, as input arguments.

Each node in the CFG represents a single source code line and edges between the nodes represent flow of control between the lines of code. Assume a change was made to line 2 (node n_0) in Fig. 1 where the comparison operator is changed from $==$ to $<=$. Node n_0 in Fig. 1 shows line 2 after the change is made to the program. The shaded nodes in the CFG represent lines impacted by the change at node n_0. DiSE explores and characterizes only the seven path conditions (execution paths) impacted by the change at line 2. Full symbolic execution of the changed version explores the entire symbolic state space, generating 21 path conditions. As a result, any client analysis which uses the results of full symbolic execution may unnecessarily analyze program behaviors that are not impacted by the change. Furthermore, for a small example such as this, a full analysis is feasible; however, for larger methods or when complex constraints are involved, a full analysis may be intractable.

C. Directed Incremental Symbolic Execution

DiSE uses information about syntactic program changes coupled with the results of data- and control- flow analyses to direct symbolic execution. DiSE generates impacted path

conditions and avoids generating path conditions that are not impacted by the changes. Path conditions generated by DiSE encode program execution behaviors impacted by the changes to the code.

1) Inputs to DiSE: The inputs to DiSE are the source code for two versions of a procedure in programs P and P', and the results of a lightweight syntactic *Diff* analysis comparing the source code for P and P', e.g., textual or abstract syntax tree comparison. The results of the *Diff* analysis identify the change set – the set of locations in the source code that are different between the two versions. For program P', the *Diff* analysis marks source code lines that are *added*, *changed*, or *unchanged* with respect to P. Similarly, the analysis marks source code lines as *removed*, *changed*, or *unchanged* in P with respect to P'.

2) Static Impact Analysis: DiSE uses standard intraprocedural program slicing techniques, with the initial change set as the slicing criteria, to generate the set of program locations that may be impacted by the actual changes. Consider the CFG computed for the modified version of `update` shown in Fig. 1. Each node in the CFG corresponds to a program location in the source code. Node identifiers appear in *italics* just outside the node, e.g., n_1, n_2. Edges between the nodes represent the possible flow of execution between the program statements.

DiSE uses the change set computed by the *Diff* analysis to mark n_0 as *changed* and then performs program slicing to identify program statements at n_1, n_3, n_4, n_5, n_{11}, n_{13}, and n_{14} as impacted write statements—the values written at these locations may impact subsequent execution of conditional branch statements. The conditional branch statements at n_0, n_2, n_{10}, and n_{12} are also identified as impacted—statements that may be impacted by the change and that may in turn affect the path condition.

3) Directed Symbolic Execution: To compute the impact of the changes on the execution behaviors of the modified version of `update`, the impact set (of program locations) computed by the static impact analysis is used to direct symbolic execution of the modified version of `update` and generate impacted path conditions. To illustrate how DiSE prunes symbolic execution using the impact set, consider a feasible execution path, $p_0 := \langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{11} \rangle$, generated during directed symbolic execution. The path p_0 contains the sequence of impacted nodes, $\langle n_0, n_1, n_5, n_{10}, n_{11} \rangle$, and the sequence of nodes $\langle n_6, n_7 \rangle$ that is not impacted by the change. However, another feasible path, $p_1 := \langle n_0, n_1, n_5, n_6, n_8, n_9, n_{10}, n_{11} \rangle$, is *pruned* (not explored) during symbolic execution because the sequence of *impacted* nodes is already covered by p_0. The only difference between p_0 and p_1 is the sequence of nodes that is not impacted by the change.

D. Limitations of an Intraprocedural Change Impact Analysis

DiSE is an intraprocedural analysis that does not account for the flow of impact between methods. Most realistic programs consist of multiple methods passing data from one method to another following call chains. Information in interprocedural programs flows forward from one method to another through the use of method arguments while information flows back to the calling method through values returned by the callee. Similarly, when a change is made to a method, the impact of the change can flow to other methods through arguments and return values. Because DiSE does not account for the flow of impact between methods, it will not generate the full set of impacted behaviors. To illustrate, consider two methods: `A` and `B`.

```
int A (int x) {x = x+1; return B(x);}
int B (int x) {if (x>0) return 1;
               else return 0;}
```

Method `A` contains an assignment to an input variable x and then passes x as an argument to method `B`. Method `B` returns either 0 or 1 based on the value of x. Suppose, a change is made to the assignment statement and now $x = x - 1$; the intraprocedural static impact analysis is *unable* to mark conditional statements in `B` as impacted by the change in method `A`. As result, DiSE will miss exploring one (either $x - 1 > 0 \wedge ret = 1$ or $x - 1 <= 0 \wedge ret = 0$) of the impacted program behaviors. To account for the flow of impact between the methods in a program, an *interprocedural* impact analysis technique is necessary.

III. Interprocedural DiSE

In this section we present an extension to the DiSE technique described in Section II that can be used to analyze the impact of changes to program behaviors across methods. We refer to this approach as interprocedural DiSE or iDiSE. The iDiSE algorithm first performs a static approximation of the impact sets across the calling contexts of methods in the program and annotates this information on a call graph. During symbolic execution the impact set information is dynamically refined using the current calling context. Combining static and dynamic information in iDiSE enables efficient generation of impacted program behaviors across multiple methods.

A. Static Impact Analysis

In phase I of the iDiSE algorithm, the static impact analysis uses as input the source code of a program P' and the differences between programs P and P' to generate a set of impacted program locations for P'. The impact analysis generates impact sets for each method in P'. A call graph is first constructed to capture the possible call sequences between methods in the program. For each method in the call graph, an impact set is generated for the method itself. The potential flow of change impact information between methods is stored by annotating the edges in the call graph and generating additional impact sets for the formal parameter of each method.

1) Call Graph Construction: The iDiSE static impact analysis first generates a call graph for P'. The call graph contains one node for every method in P'. The edges between the nodes in the call graph represent a method invocation (caller $\rightarrow$ callee). Each node in the call graph has the same number of outgoing edges as the number of method invocations. If

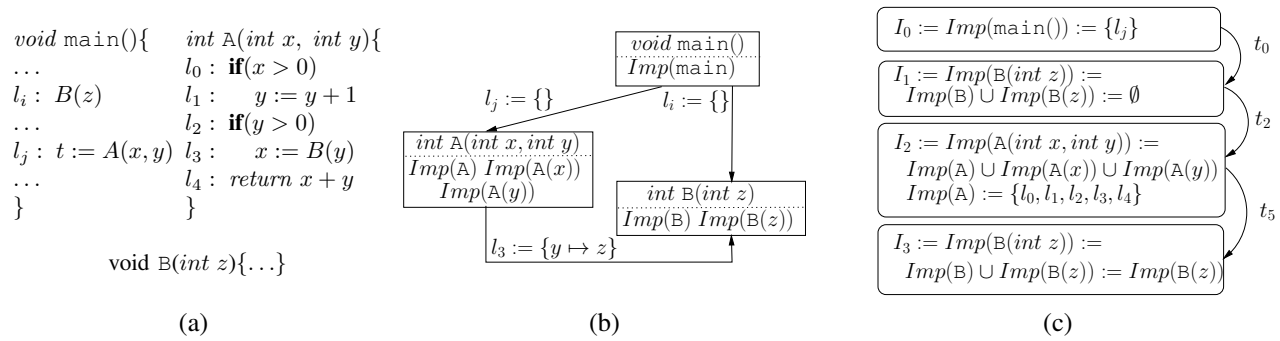

Fig. 2. Example to demonstrate the use of static and dynamic calling context in iDiSE. (a) An interprocedural program code snippet. (b) Call graph with impact sets for each method and static calling context. (c) The impact sets dynamically refined based on the calling context.

method, m_j, is invoked twice from method, m_i, then there are two edges from m_i to m_j in the call graph. Each edge in the call graph is labeled with the program location of the method invocation site. A call graph corresponding to the program in Fig. 2(a) is shown in Fig. 2(b). The example in Fig. 2 contains three methods: `main`, `A`, and `B`. In `main` there is an invocation to `B` at location l_i followed by an invocation to `A` at location l_j.

2) Generating an Impact Set: Impact sets for each method are generated using standard program slicing techniques as described in detail in [7]. High-level pseudocode for the dependence analysis used to generate the impact sets is shown in Fig. 3(a). The impact set, *Imp* for a method, m, is initialized with the changes to the method. The impact set, *Imp*, is provided as input to each of the procedures, `DataFlow`, `ControlFlow`, and `CallFlow` in Fig. 3(a) until a fixpoint is reached.

In the `DataFlow` procedure shown in Fig. 3(a), line 1 states that if at impacted location l_a a value is assigned to a variable ($\texttt{Def}(l_a)$) which may be used at another program statement l_b ($\texttt{Uses}(l_b)$), then at line 2, l_b is added to the set of impacted program statements. Suppose, the program statement $l_1 : y := y + 1$ in Fig. 2(a) is the single element of the change set for method `A` computed by the *diff* analysis and provided as input to iDiSE. The value assigned to y at line l_1 may be used in the conditional statement at l_2, as the argument to method `B` at line l_3, and in the return value of $x + y$ at line l_4; hence, there is a forward impact from l_1 to l_2, l_3, and l_4.

In the data flow analysis we also consider backward data flow. In Fig. 3(a), line 6 states that if impacted location l_b uses a value ($\texttt{Uses}(l_b)$) assigned at location l_a ($\texttt{Def}(l_a)$), then at line 7, location l_a is added to the impact set. Suppose, two assignments to variable x, $x = 0$ and $x = 10$ can reach an impacted statement, **return** $x + 1$ (changed from **return** $x - 1$). We would miss impacted behaviors one of the outputs ($x = -1$ changed to $x = 1$ or $x = 9$ changed to $x = 11$) if we do not consider both assignments to x that flow to the impacted return statement.

We also consider control flow dependence between program statments. Line 10 in Fig. 3(a) checks if impacted location l_a is control dependent on a conditional statement l_b, then at line 11 the program statement l_b is added to the impact set. In Fig. 2(a), the execution of l_1 is control-dependent on the execution of the $true$ branch of the conditional statement at l_0. This introduces a backward dependence edge between l_1 and l_0.

To compute interprocedural results, the impact analysis also marks impacted call sites. A call site is marked as impacted if the invocation leads to a method with a non-empty change set through some path in the call graph. In Fig. 3(a), in the `CallFlow` procedure at lines 14 and 15, if a call site, l_a, in method m invokes another method m' such that the impact set of m' is non-empty, then the call site, l_a, is added to the impact set. For the example shown in Fig. 2(a), program location, l_j, in the method `main` is marked as impacted by iDiSE because it leads to method `A` which has a non-empty impact set.

The impact set for method `A` in Fig. 2(a) for the change set of $\{l_1\}$ is $\{l_0, l_1, l_2, l_3, l_4\}$. Standard program slicing techniques using the dependence analysis described in this section are used to generate the impact set for a given change set.

3) Impact Sets for Formal Parameters: Change impact information flows from a caller to a callee through one or more arguments in the method invocation. The formal parameters in the callee method are marked as impacted based on the caller invoking the method. It is possible that in some contexts, the impact of a change will not flow through to the callee. Analyzing a method in the context of multiple invocation sites has the potential to cause the analysis to generate an exponential number of impact sets. To avoid this, iDiSE computes a separate impact set for each formal parameter of the method; iDiSE analyzes each method only *once*.

The total number of impact sets generated for a method is the number of formal parameters plus one for the change impact within the method body. We refer to the latter as the impact set for the method. The `main` method in Fig. 2(a) does not have formal parameters so it has a single impact set `Aff(main)` as shown in Fig. 2(b). Whereas, method `A` has two formal parameters, resulting in a total of three impact sets. One for the method itself ($Imp(\texttt{A})$) and one for each of its formal parameters ($Imp(\texttt{A}(x))$ and $Imp(\texttt{A}(y))$).

To generate the impact set for a formal parameter, the

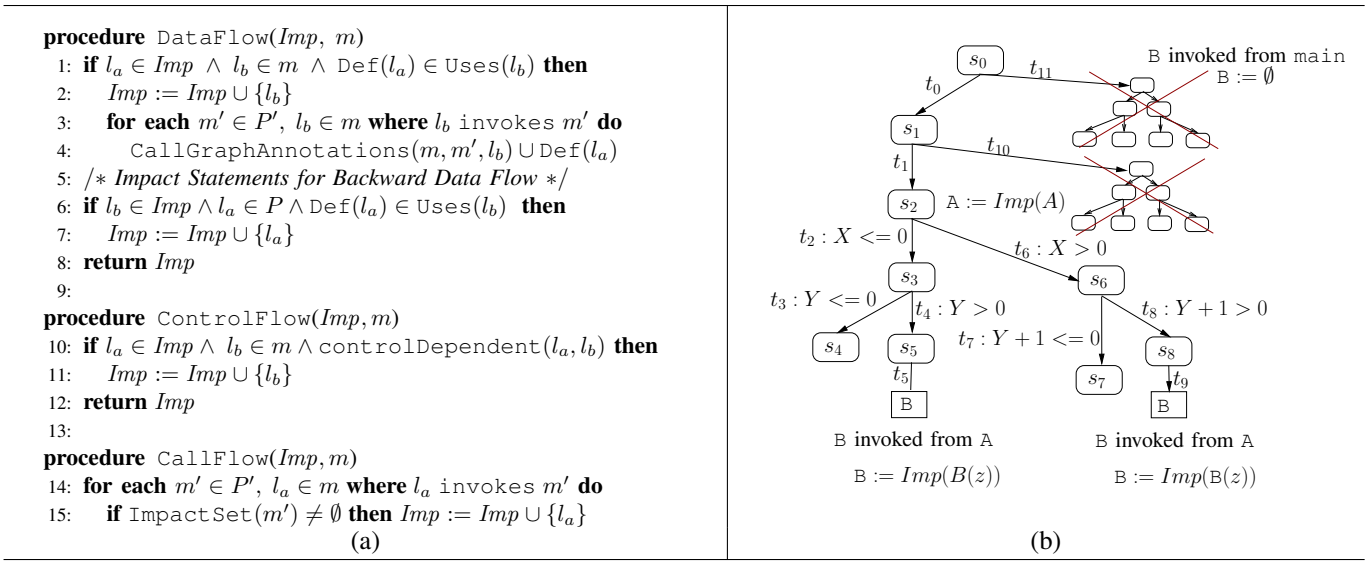

Fig. 3. (a) Pseudocode for the dependence analysis used to compute the impact sets in iDiSE. (b) Partial Symbolic Execution Tree for the code in Fig. 2(a).

change set is initialized to the program locations that *use* the impacted formal parameter. The initial change set for the impacted formal parameter *int* x in method A is $\{l_0, l_4\}$ because the conditional statement at l_0 and return statement at l_4 are the only statements that use (read) the variable x.

4) Annotating Call Edges: Edges in the call graph are annotated using the argument names in the caller and their corresponding formal parameter names in the callee to represent the flow of change impact information from caller to callee. At lines 3 and 4 in Fig. 3(a), the CallGraphAnnotations method adds argument names along a call edge when m' is invoked from m at location l_b. The arguments annotated are the ones that are assigned a value at an impacted location l_a (Def(l_a)). The change at line l_1 in method A shown in Fig. 2(a) impacts the call site at l_3. Variable y is assigned a value at l_1 and is passed as an argument when invoking method B at l_3. The edge from A to B in Fig. 2(b) is annotated with $l_3 : y \mapsto z$ which indicates the flow of the impact of the change to y in A to B, and also maps the argument y to the formal parameter, z, in method B.

B. Directed Symbolic Execution

During directed symbolic execution, paths are pruned based on the reachability of impacted program locations from the current state. Before checking reachability, iDiSE dynamically refines the impact sets for a given method based on which caller invoked the method and the annotation on the corresponding call edge in the call graph. This approach efficiently generates precise change impact information.

1) Pruning Example: A partial symbolic execution tree is shown in Fig. 3(b). Execution begins at the initial state, s_0, generating the next symbolic states s_1 and s_2, where s_2 is a successor of s_1 on the current execution path. Assume states s_1 and s_2 are generated by execution of program statements in method B when it is invoked from main. Next, we see in Fig. 3(b), states labeled s_3 and s_4 are generated respectively by transitions $t_2 : X <= 0$ and $t_3 : Y <= 0$, on the current path after which method A returns. At this point, symbolic execution backtracks to generate the other choice of s_5 and then explores B. An impact set ($B := Imp(B(z))$) for method B is dynamically generated when transition t_5 is executed. The information about impacted program statements in this set is used to direct execution within method B. Eventually, when the search backtracks to states s_1 and s_0 in Fig. 3(b), the transitions t_{10} and t_{11} lead to subtrees generated by executing instructions in method B. Subtrees reachable from t_{10} and t_{11} are pruned because the impact set of B is empty when invoked from main. The fact that the impact set for B is empty indicates that the transitions, t_{10} and t_{11} do not lead to execution paths that are impacted by the changes. The details on dynamically generating impact sets and checking reachability to prune paths is described in detail below. The goal of Fig. 3(b) is to demonstrate to the reader that the iDiSE algorithm uses information from the static impact analysis coupled with the actual calling context to decide *when* to explore paths in a certain method. When method B is invoked from from A, the iDiSE analysis generates paths through method B; however, the paths through B are pruned in the iDiSE analysis when B is invoked from main.

2) Dynamically Generating Impact Sets: When a method is invoked during symbolic execution, the final impact set for the method is dynamically generated based on the method's calling context and information computed by the static analysis. For the example in Fig. 2(a), the dynamically generated impact sets for the methods are shown in Fig. 2(c). When method B is invoked from main, its final impact set, I_1, is empty for two reasons. First, the impact set for B itself is empty. Second, there are no annotations on the call edge from main to B in Fig. 2(b); hence, the elements from the impact set for the formal parameter z of method B are *not added* to the final set. When, however, method B is invoked from method

A, the impact set of its formal parameter z—$Imp(B(z))$—is assigned as the final impact set of B as shown in the set labeled I_3 in Fig. 2(c). Dynamically generating the final impact sets based on the calling context of the method allows iDiSE to efficiently track the impact of the flow of changes between different methods.

Dynamically generated impact sets for a method have a *scope* within which they can be used to check reachability. Fig. 2(c) is a list of impacted sets in scope when transition t_5 in Fig. 3(b) executes. The impact set I_3 is in scope for all transitions in method B following t_5. When the symbolic execution backtracks to state s_2, to a point before the invocation of B, the impact set I_3 is removed from the list of impact sets in scope. When transition t_9 in Fig. 3(b) is executed, method B is invoked again from method A. At this point a new final impact set, I_4, is generated for method B and added to the list of impact sets in scope: $\langle I_0, I_1, I_2, I_4 \rangle$; contents of I_4 are identical to those of I_3. Eventually, when the search backtracks to state s_1 in Fig. 3(b), the sets in scope are $\langle I_0, I_1 \rangle$. Here reachability to impacted statements within method B is checked using I_1. Checking reachability to impacted statements is described next.

3) Checking Reachability: During symbolic execution, paths are explored or pruned based on reachability from the current symbolic state to the program instructions in the impacted set. Our previous work on DiSE checks reachability to impacted program locations with respect to a particular impact set. It maintains separate sets of explored and unexplored impacted program locations within a method. In iDiSE (and DiSE), if an impacted program location, l', in the unexplored set is reachable from the current program location, l, then execution continues along that path otherwise the path and its corresponding sub-tree is pruned. Reachability between l and l' is checked using the CFG for the method—does there exist a sequence of edges from l to l' in the CFG? The details of the reachability check are described in [7].

The reachability check in iDiSE is extended to account for reachability of program statements in other methods. In iDiSE, we unroll the call stack to check reachability in other methods within the current calling context. After checking reachability within the current method, iDiSE uses the call stack to extract the sequence of method invocations that led to the current method. A possible calling context is as follows: main $\rightarrow$ A $\rightarrow$ B for the example shown in Fig. 2(b). In this context we want to check whether an execution path can be pruned while the execution is at a location in method B. Suppose there are no unexplored impacted program locations in method B that are reachable from the current location, then we look at the call site from A that invoked the current instance of method B. We then check if there exists a unexplored impacted program location in method A reachable from the call site. The check continues along the call chain until we get through the series of invocations or we find a reachable, unexplored, impacted program statement.

Dynamically generating impacted sets and computing reachability along the calling context of the program allows iDiSE to efficiently prune parts of the symbolic execution tree that are not impacted by the changes.

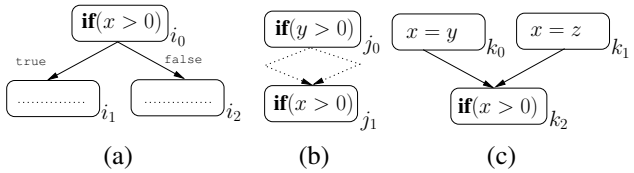

Fig. 4. Different notions of coverage for impacted program locations: (a) impacted branch coverage and basic block coverage, (b) impacted path coverage, (c) impacted data flow coverage.

IV. Analysis configurations

Traditional notions of coverage are insufficient for characterizing the test effort required to validate evolving programs because they do not distinguish between coverage of impacted and unimpacted parts of the program. In this section, we describe how traditional notions of coverage are extended to incorporate information about change impact. We describe how these notions of impacted coverage can be used to configure techniques such as DiSE and iDiSE when applied to regression testing. We believe these notions of impacted coverage are generic and can be applied to the evaluation of a wide variety of change impact analysis and regression testing techniques. This section also presents a discussion on how DiSE and iDiSE can be configured for debugging—finding the root cause of errors introduced by a change.

A. Impacted Coverage Metrics

Traditional notions of code coverage, e.g., statement or branch coverage, are useful for describing whether or not parts of the code have been executed during, for example, regression testing; however, when used in the context of evolving software, these coverage metrics do not provide any indication regarding the coverage of *impacted* statements and branches – valuable information for evaluating the results of software maintenance tasks such as regression testing. In this work, we extend several traditional notions of code coverage to incorporate change impact information. These new notions of coverage are useful for configuring DiSE and iDiSE, and for relating the behavioral coverage computed by directed symbolic execution, and described by the path conditions, back to the source code.

The CFGs shown in Fig. 4 illustrate the impacted coverage notions defined in this work. CFGs provide an intuitive mechanism for illustrating the impacted coverage notions because each constraint on a path condition generated by symbolic execution can be directly mapped to a conditional branch in the code (and in the CFG). In Fig. 4, each node in a CFG represents a single program statement.

Definition IV.1. Impacted Statement Coverage: *Cover every impacted program statement.*

Definition IV.2. Impacted Basic Block Coverage: *Cover every basic block that contains an impacted program statement.*

Impacted statement coverage entails covering every executable program statement that is impacted by the change. Impacted basic block coverage is very similar to impacted statement coverage. Because many analyses are performed at the basic block level, rather than the program statement level, it is important to define coverage metrics for both. Suppose the conditional branch statement at node i_0 in Fig. 4(a) is marked as impacted. When the `ImpactedStatement` or `ImpactedBasicBlock` option is specified, the analysis will generate path conditions through nodes i_0 to i_1 *or* through nodes i_0 to i_2; covering the impacted statement i_0. Here, i_0 is also an impacted basic block.

Definition IV.3. Impacted Branch Coverage: *Cover every possible decision of impacted control structures (`if`-statements, `while`-statements, `switch`-statement cases), impacted exception handlers, and all impacted entry and exit points.*

When DiSE or iDiSE is configured with the option, `ImpactedBranch`, the analysis generates feasible path conditions through the $true$ and $false$ branches of all *impacted* conditional branch statements. The analysis also generates path conditions for the various cases in *impacted* switch statements, impacted exception handlers, and impacted entry and exit points, e.g., entry and exit points of an impacted method. When `ImpactedBranch` is specified for the example in Fig. 4(a), the analysis generates path conditions through nodes i_0 to i_1 *and* through nodes i_0 to i_2.

Definition IV.4. Impacted Path Fragment Coverage*: Cover every possible combination of impacted path fragments, where an impacted path fragment is a sequence of program statements that ends in an impacted conditional statement and the path fragment contains only one impacted conditional statement.*

Each program execution path is composed of one or more path fragments. Different combinations of path fragments result in different paths. Similarly, different combinations of impacted path fragments result in different impacted paths. Suppose, the conditional branch statements at j_0 and j_1 in Fig. 4(b) are both impacted. The sequence of edges leading to j_0 and the sequence of edges leading to j_1 each constitute a path fragment. We have shown the sequence of edges as a dotted edge in Fig. 4(b) for brevity. Using the `ImpactedPathFrag` option, DiSE or iDiSE will generate the following paths (path conditions): (1) $x > 0 \wedge y > 0$, (2) $x > 0 \wedge y \leq 0$, (3) $x \leq 0 \wedge y > 0$, and (4) $x \leq 0 \wedge y \leq 0$.

Definition IV.5. Impacted Data Flow Coverage *: Cover every use of a variable defined at an impacted write statement.*

The goal of impacted data flow coverage is to ensure every possible use of every definition in an impacted write statement is covered. Suppose, the assignments to x at nodes k_0 and k_1 are both impacted in Fig. 4(c). In this example x is subsequently used in the conditional branch statement at k_2. When DiSE or iDiSE is configured with `ImpactedDataFlow`, it generates four path conditions: $y > 0$, $y \leq 0$, $z > 0$, and $z \leq 0$. The impacted data flow coverage obligations can then be fulfilled by solving these path conditions.

B. Debugging

Changes to programs can impact the correctness of the program by introducing unexpected program behaviors (errors or bugs). Regression testing techniques often allows us to detect these errors, but, they do not provide information about the root cause of errors. Techniques such as DiSE and iDiSE can be configured to facilitate the process of finding the cause of an error (debug it).

The DARWIN method for debugging evolving programs detects potential conditions for failing test cases when a change is made to a program [13]. Suppose a test case T passes in P, but, fails in P'. Darwin generates a path condition, $\Phi := \phi_0 \wedge \ldots \wedge \phi_n$, in P along the path which is generated by running T; and similarly generates $\Phi' := \phi'_0 \wedge \ldots \wedge \phi'_m$ in P'. Next, for each ϕ'_i where $0 \leq i < m$, the DARWIN method computes a new formula $\Phi_i := \Phi \wedge \phi'_0 \wedge \ldots \wedge \neg\phi'_i$; if the new formula Φ_i is satisfiable then DARWIN reports the location where ϕ'_i is generated as a potential cause for the failing test.

DiSE and iDiSE results can be used to improve on the DARWIN debugging approach by configuring DiSE (or iDiSE) with an option `DeltaDebug` to retain only the constraints generated at impacted conditional branch statements. This approach safely reduces the number of constraints on the path conditions and enables the debugging process to focus on impacted program locations as the potential cause for failing test cases.

V. Evaluation

We now present the results of a study in which we evaluate the effectiveness of our change impact analysis technique.

A. Tool Support

We have implemented DiSE and iDiSE in the Java PathFinder symbolic execution framework (SPF) [11], [12]. We use the Choco constraint solver [15] to check path feasibility during symbolic execution. Our analysis uses a custom interprocedural data- and control-flow analysis to compute a conservative approximation of the impacted program statements. The artifacts in our study were compiled with Java version 1.6.0_26. We then computed a source-level diff using a custom Abstract Syntax Tree (AST) differencing application written in Java.

B. Artifacts

To evaluate our analysis, we used two Java artifacts that are representative of programs used in the evaluation of symbolic execution techniques. The first, `tcas`, is an aircraft collision avoidance system from the Software-artifact Infrastructure Repository (SIR) [16]. SIR contains 41 versions of `tcas`, each of which is a mutant of version 0 containing one or

Version (P-P')	Time (P')		Time (P)		States (P')		States (P)	
	Full	iDiSE	Full	iDiSE	Full	iDiSE	Full	iDiSE
V0V2	0:01:29	0:00:48	0:01:30	0:00:48	679	536	679	536
V0V3	0:01:47	0:00:02	0:01:30	0:00:02	837	32	679	21
V0V4	0:01:36	0:00:06	0:01:30	0:00:08	743	216	679	208
V0V5	0:01:49	0:00:02	0:01:30	0:00:02	763	26	679	31
V0V6	0:01:41	0:01:01	0:01:30	0:00:57	679	560	679	560
V0V10	0:01:33	0:01:36	0:01:30	0:01:43	775	759	679	679
V0V23	0:01:26	0:00:11	0:01:30	0:00:50	631	96	679	536
V0V29	0:00:48	0:00:24	0:01:30	0:00:49	311	240	679	536
V0V34	0:05:58	0:00:02	0:01:30	0:00:02	2317	20	679	20
V0V40	0:01:08	0:00:02	0:01:30	0:00:02	615	31	679	32

(a)

V43V44	0:01:28	0:00:50	0:01:27	0:00:51	679	536	679	536
V45V46	0:01:34	0:01:02	0:01:35	0:00:55	695	544	695	544
V46V47	0:01:53	0:01:26	0:01:34	0:01:09	857	758	695	624
V47V48	0:02:42	0:02:57	0:01:53	0:01:53	1045	1045	857	847
V49V50	0:02:53	0:00:54	0:03:01	0:01:19	1435	980	1205	846
V50V51	0:02:49	0:00:04	0:02:53	0:00:04	1435	44	1435	44

(b)

TABLE I
COSTS OF RUNNING IDISE VS. FULL SYMBOLIC EXECUTION FOR TCAS.

two functional changes (e.g. operator mutations, modified constants, and modified control structures). We translated the C versions in SIR to Java. Each version has 10 methods and contains 185 SLOC. We created an additional ten versions (42 – 51) to simulate a software evolution process where each version is based on the previous version, e.g., version 43 is a mutant of version 42. We used version 0 as our base version. Versions 42 – 44 contain refactoring changes, i.e,. changes that do not affect the functionality of the program, and versions 45 – 51 contain functional changes.

To illustrate the analysis configurations discussed in Section IV, we use the Wheel Brake System (WBS) example, a synchronous reactive component from the automotive domain. The Java model consists of one class and 231 SLOC. It is based on a Simulink model derived from the WBS case example found in ARP 4761 [17], [18]. We evaluate the `update` method in WBS.

C. Results and Discussion

Costs of iDiSE: *How do the costs of iDiSE compare to full symbolic execution in terms of time and states explored?* In Table I, we report the total *wall clock time* in seconds and the *states explored* during directed symbolic execution for iDiSE and for full symbolic execution. The impacted program behaviors for each pair of `tcas` versions, P and P', are generated by iDiSE using the *DeltaDebug* configuration described in Section IV-A. For the `tcas` artifact, iDiSE is able to achieve significant reductions in terms of the total time and states explored as shown in Table I. Versions V0V29 and V0V34 in Table I(a) require the minimum and maximum analysis times respectively for full symbolic execution from the set of 41 versions in SIR. Versions V0V5 and V0V10 have the minimum and maximum analysis times respectively for iDiSE from that same set. Similar reductions in time and states explored were noted in all 41 versions of `tcas` and also in the evolutionary versions of `tcas` shown in Table I (b).

Benefits of iDiSE: *What reductions can be achieved by iDiSE?* Table II shows that, in general, iDiSE generates fewer path conditions (PCs) than symbolic execution for `tcas`. In V0V23, iDiSE generates 3 PCs in program P', and 29 PCs in P, whereas full symbolic execution generates 866 PCs for P' and 595 PCs for P. Similar reductions can be observed for the other versions of `tcas`.

Because of space limitations, Table I and Table II contain a representative subset of the version pairs evaluated. Nevertheless, these results demonstrate that iDISE is an effective technique for analyzing evolving software because it can generate impacted behaviors and avoid exploring behaviors that are not impacted by the change. Moreover, because the set of impacted behaviors computed by DiSE and iDiSE is reduced with respect to full symbolic execution, the client analysis that uses the results will also benefit. For example, to check for input partition equivalence [14] for V0V2 in Table II, it is necessary to compare only the 29 path conditions in P and P', rather than the 595 path conditions computed by full symbolic execution. This is one of the key strengths of our technique.

In Table II, we also report the number of constraints associated with impacted branch conditions (*Impacted Constraints*) versus unimpacted branch conditions (*Unimpacted Constraints*). This information, along with the percentage of the total constraints, is useful for debugging-related client analyses. For example, in V0V2 there are 216 constraints generated at conditional statements not impacted by the changes, while 319 constraints are generated at impacted conditional statements. In this case, approximately 40% of the constraints generated during directed symbolic execution are not impacted by the change. In other versions of the `tcas` example, often a considerable percentage of the constraints are generated at conditional branches not impacted by the change. As a result, a client analysis such as the DARWIN debugging approach described in Section IV can be improved using iDiSE results to detect program locations that are potential causes in failing tests. Checking fewer constraints can reduce the cost of locating potential causes of failing tests. Furthermore, the quality of the potential causes reported can be improved because only impacted locations are reported.

Benefits of Configurability: *What are the costs and benefits of using impacted coverage configurations?* The coverage configurations are technique-agnostic; they can be used by both DiSE and iDiSE, and other change impact analyses. In Table III we present the number of states generated, total wall clock time, and path conditions generated when running DiSE on the WBS example configured for several notions of impacted coverage defined in Section IV—impacted branch coverage (Branch), impacted basic block coverage (Basic Block), and impacted path fragment coverage (Path Frag.). We also present the total number of path conditions generated by full symbolic execution (Full). In general, the different configurations of DiSE generate fewer path conditions compared to full symbolic execution. In version V0V4, DiSE generates only a single path condition for covering the impacted branches, basic blocks, and path fragments;

whereas full symbolic execution generates 24 path conditions because it explores the entire symbolic state space. The results in Table II and Table III illustrate the configurability of DiSE and iDiSE and show the tradeoff in cost and precision for these configurations.

Note that we do not attempt to compare DiSE and iDiSE directly; however, we note that DiSE is useful as an inexpensive, local analysis of a method. If a change is made to an algorithm of a single method in a large system, DiSE can pinpoint the analysis to that method, avoiding the expense of a system-wide analysis. For programs with multiple methods, it is essential to use iDiSE so that the effects of the changes can be propagated between methods.

The threats to validity in our study include (1) the tools upon which our technique is built, e.g., SPF, JPF, Choco, (2) the selection of artifacts used to evaluate the benefits of iDiSE, and (3) the changes applied to create the mutants. Implementing iDiSE in another framework or using another constraint solver/decision procedure could produce different results; however, replicated studies with other tool frameworks would address this threat. The artifacts selected for our study are control applications that are amenable to symbolic execution. They are comparable in structure and complexity to other artifacts that we are aware of that are used to evaluate symbolic execution techniques. The mutant versions in the WBS example were created manually, and may or may not reflect actual program changes; however, the mutations were developed in a systematic way that considered program location, change type, and number of changes. The tcas example was chosen because it is a control system with a version history.

VI. Related Work

The technique presented in this paper, iDiSE, combines static program slicing and symbolic execution to estimate the impact of changes made to software. The results of iDiSE characterize the impact in terms of program execution behaviors. iDiSE is an extension of our previous work in [7] that now includes an interprocedural analysis and options that enable the impact analysis to be configured to support the requirements of specific client analyses that use the DiSE and iDiSE results. The techniques most closely related to our work are Differential Symbolic Execution (DSE) [14] and Regression Model Checking (RMC) [19]. Like our technique, DSE utilizes symbolic execution to characterize the effects of program changes; however, DSE does not use program slicing techniques to improve the efficiency of symbolic execution. It instead relies on abstract summaries of unchanged code blocks. RMC uses differences between two versions of a program to drive pruning of the state space during model checking of the new version of the program. Similar to DiSE and iDiSE, RMC uses a static impact analysis to calculate 'dangerous' elements whose behavior may be impacted by the changes.

Many techniques have been developed to assist developers maintain evolving software. Tools that difference various representations of program source code [20], [21], [22], [23] are used regularly by practitioners to detect changes. Determining the impact of the changes is more challenging however, both in terms of detecting what is impacted and in how the impact set is characterized and reported. Dynamic impact analysis techniques [5], [6] use information collected during execution of the program, however, the impact sets computed by such techniques are restricted to observed behaviors. By using symbolic execution to compute impact sets, our technique has the potential to analyze more observed behaviors; however, the limitations of symbolic execution may also limit the capabilities of our technique unless the limitations are manifested in the parts of symbolic execution pruned by DiSE and iDiSE. Reporting change impact sets in terms of the code impacted [4], [24], [25] or test cases impacted [3], [26] provides developers with a mechanism for targeting the impact of the changes during subsequent evolutions tasks, e.g., regression testing. Because our technique uses symbolic execution, the impact sets reported to the user are characterized in terms of path condition, representing the effects of changes on program execution paths. Another recent work, [27], uses program slicing based on program outputs, and backwards symbolic execution to reason about program changes.

VII. Conclusions and Future Work

In this paper, we illustrate how the results of our change impact analysis technique, iDiSE, can be used to support software evolution tasks, including testing and debugging. We describe novel configuration options that combine traditional notions of coverage with change impact information, and explain how the impact analysis performed by DiSE and iDiSE can be configured to support the specific requirements of the client analysis. We also introduce the algorithms supporting the interprocedural version of our analysis, iDiSE and demonstrate that like DiSE, the cost of generating impacted program behaviors using interprocedural DiSE, is less than the cost of full symbolic execution. In the future, we plan to conduct a more comprehensive evaluation comparing how the results of iDiSE can be used to improve the efficiency of client analysis techniques relative to other state-of-the-art techniques.

References

[1] R. S. Arnold and S. A. Bohner, "Impact analysis - towards a framework for comparison," in *ICSM '93*, 1993, pp. 292–301.

[2] S. A. Bohner, "Software change impact analysis," 1996.

[3] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip, "Chianti: a change impact analysis tool for Java programs," in *ICSE*, 2005, pp. 664–665.

[4] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *ICSE*. New York, NY, USA: ACM, 2011, pp. 746–755.

[5] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 308–318.

[6] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *ICSE*. New York, NY, USA: ACM, 2005, pp. 432–441.

[7] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *PLDI*, 2011, pp. 504–515.

[8] L. A. Clarke, "A program testing system," in *Proceedings of the 1976 annual conference*, ser. ACM '76, 1976, pp. 488–491.

[9] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

Version (P-P')	Path Conditions (P')		Path Conditions (P)		Impacted Constraints		Unimpacted Constraints	
	Full	iDiSE	Full	iDiSE	iDiSE (P')	iDiSE (P)	iDiSE (P')	iDiSE (P)
V0V2	595	29	595	29	319	319	216	216
V0V3	706	2	595	3	27	16	4	4
V0V4	783	21	595	21	183	175	32	32
V0V5	823	3	595	4	21	24	4	6
V0V6	595	37	595	37	431	431	128	128
V0V10	783	84	595	68	438	406	320	272
V0V23	866	3	595	29	75	439	20	96
V0V29	321	25	595	29	183	319	56	216
V0V34	2257	2	595	2	19	19	0	0
V0V40	587	2	595	2	28	29	2	2

(a)

V43V44	595	29	595	29	319	319	216	216
V45V46	713	29	713	29	431	431	112	112
V46V47	938	60	713	53	665	527	92	96
V47V48	1257	133	938	89	720	620	324	226
V49V50	2749	98	1801	54	819	685	160	160
V50V51	2749	10	2749	10	43	43	0	0

(b)

TABLE II
PATH CONDITIONS GENERATED BY IDISE AND FULL SYMBOLIC EXECUTIONS FOR THE `tcas` EXAMPLE.

Version (P-P')	Time			States			Path Conditions			
	Basic Block	Branch	Path Frag.	Basic Block	Branch	Path Frag.	Full	Basic Block	Branch	Path Frag.
V0V1	0:02:50	0:00:03	0:00:08	663551	55	5715	24	24	1	6
V0V2	0:00:04	0:00:03	0:00:03	48	45	45	24	18	17	17
V0V3	0:00:20	0:00:19	0:00:19	64091	64091	64091	24	12	12	12
V0V4	0:00:04	0:00:04	0:00:04	13	13	13	24	1	1	1
V0V5	0:00:17	0:00:17	0:00:16	58979	58979	58979	24	14	14	14
V0V6	0:00:04	0:00:03	0:00:03	13	13	13	24	1	1	1
V0V7	0:02:50	0:00:04	0:00:08	663551	55	5715	24	24	1	6
V0V8	0:00:18	0:00:17	0:00:18	58979	58979	58979	24	14	14	14
V0V9	0:00:20	0:00:19	0:00:19	64091	64091	64091	24	12	12	12
V0V10	0:02:48	0:00:03	0:00:08	663551	55	5715	24	24	1	6
V0V11	0:00:17	0:00:18	0:00:16	58979	58979	58979	24	14	14	14
V0V12	0:00:17	0:00:17	0:00:19	58979	58979	69243	24	14	14	6
V0V13	0:00:16	0:00:17	0:00:18	58979	58979	58979	24	14	14	14
V0V14	0:02:49	0:00:04	0:00:08	663551	55	5715	24	24	1	6
V0V15	0:02:48	0:00:20	0:00:20	663551	64099	64099	24	24	12	12
V0V16	0:00:17	0:00:16	0:00:17	58979	58979	58979	24	14	14	14

TABLE III
DISE WITH VARIOUS IMPACT COVERAGE CONFIGURATIONS FOR `WBS`.

[10] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs." *ASE*, vol. 10, no. 2, pp. 203–232, 2003.

[11] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *ISSTA*, 2008, pp. 15–25.

[12] C. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *ASE*, 2010, pp. 179–180.

[13] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: an approach for debugging evolving programs," in *ESEC/FSE 2009*. New York, NY, USA: ACM, 2009, pp. 33–42.

[14] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *FSE*, 2008, pp. 226–237.

[15] Choco, "Main–page Choco," —http://www.emn.fr/z-info/choco-solver/—, 2010.

[16] SIR, "Software-artifact infrastructure repository: Home," —http://sir.unl.edu—, 2008.

[17] SAE-ARP4761, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[18] A. Joshi and M. Heimdahl, "Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier," in *SAFECOMP*, ser. LNCS, vol. 3688, September 2005, pp. 122–135.

[19] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *ICSM*, 2009, pp. 115–124.

[20] T. Apiwattanapong, A. Orso, and M. J. Harrold, "JDiff: A differencing technique and tool for object-oriented programs," *ASE*, vol. 14, no. 1, pp. 3–36, 2007.

[21] Diffutils, "Comparing and merging files," http://www.gnu.org/software/diffutils, 2002.

[22] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: a semantic-graph differencing tool for studying changes in large code bases," in *ICSM*, 2004, pp. 188–197.

[23] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gail, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[24] J. Law and G. Rothermel, "Incremental dynamic impact analysis for evolving software systems," in *ISSRE '03*, 2003, pp. 430–.

[25] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proc. ESEC/FSE-11*, 2003, pp. 128–137.

[26] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "express: guided path exploration for efficient regression test generation," in *Proc. ISSTA*, 2011, pp. 1–11.

[27] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," in *ESEC/FSE '11*, 2011, pp. 278–288.