DUA-FORENSICS: A Fine-Grained Dependence Analysis and Instrumentation Framework Based on Soot

Raul Santelices, Yiji Zhang, Haipeng Cai, and Siyuan Jiang
University of Notre Dame
e-mail: {rsanteli|yzhang20|hcai|sjiang1}@nd.edu

Abstract

We describe DUA-FORENSICS, our open-source Java-bytecode program analysis and instrumentation system built on top of Soot. DUA-FORENSICS has been in development for more than six years and has supported multiple research projects on efficient monitoring, test-suite augmentation, fault localization, symbolic execution, and change-impact analysis. Three core features of Soot have proven essential: the Java bytecode processor, the Jimple intermediate representation, and the API to access and manipulate Jimple programs. On top of these foundations, DUA-FORENSICS offers a number of features of potential interest to the Java-analysis community, including (1) a layer that facilitates the instrumentation of Jimple code, (2) a library modeling system for efficient points-to, data-flow, and symbolic analysis, and (3) a fine-grained dependence analysis component. These features have made our own research more productive, reliable, and effective.

Keywords Program Dependence, Instrumentation, Library Model, Soot Analysis Framework

1. Introduction

Numerous tools have been developed over the years for the analysis and instrumentation of Java programs. Many such tools target Java bytecode, which for us is a convenient representation for three reasons: (1) it breaks down programs into manageable small instructions, (2) those instructions retain high-level object-oriented features, and (3) the source code of the program is not needed.

Only a few tools, however, offer the general infrastructure that helps developers build program analyses efficiently and reliably. Two main challenges have constrained these enabling tools. First, the bytecode instruction set evolves over time as new versions of Java are released. Second, the operand stack that the bytecode uses makes long sequences of instructions very hard to understand and correctly instrument. These challenges and, in many cases, the lack of resources for maintenance, have affected the usability of such tools. Examples are BCEL [9], JABA [6], and InsectJ [31].

The Soot Analysis Framework [17, 32], in contrast, is a well-maintained and actively supported system that provides essential features for a variety of analysis tasks. Key features of Soot that support the development of other analysis tools, in our experience, are its Java-bytecode processor, the Jimple intermediate representation, data structures for accessing and manipulating Jimple programs, and core analyses for this representation.

In this paper, we describe DUA-FORENSICS, our Java-bytecode program analysis and monitoring system built on top of these main features of Soot.[1] DUA-FORENSICS has been in development for more than six years and has supported our research on program-analysis based tasks such as efficient monitoring [22], test-suite augmentation [27], fault localization [28], symbolic execution [23], and quantitative slicing for change-impact analysis [24, 30, 34].

DUA-FORENSICS offers various features of interest to Java users and especially the Soot community. Its main features are:

1. A higher-level layer for easier instrumentation of Jimple code
2. A library-method modeling (summarizing) system for manual specification of points-to, data-flow, and symbolic effects
3. A fine-grained dependence-analysis component for detailed change-propagation analysis and program slicing

Together, these features have assured the productivity, reliability, and effectiveness of our Java-bytecode technique implementations for research. Features 1 and 2 can be particularly useful to all developers, whereas feature 3 provides advanced functionality for those who wish to investigate program dependencies in detail.

In Section 2, we introduce an example and definitions used throughout the paper. In Sections 3–6, we describe the architecture and the three main features of DUA-FORENSICS. In Section 7, we present our application of these features for a novel change-testing approach. Finally, in Section 8, we conclude and discuss common future challenges for DUA-FORENSICS and Soot.

2. Basic Definitions and Example

Figure 1(a) presents program `E`, which inputs integers `x` and `y`. In statements 2 and 4, `E` increments or decrements `y` depending on the value of `x` at statement 1. Then, `E` outputs 1 or 0 in statements 6 and 8, respectively, depending on the value of `y` at statement 5. Figure 1(a) also shows a change `ch1` in statement 1 of program `E`, described by a comment in that line. When applied to `E`, `ch1` replaces the relational operator <= with > in that statement.

Figure 1(b) shows the control-flow graph (CFG) [2] of method `E`, where each node represents a statement or the entry or exit of the method and the edges indicate which nodes succeed which other nodes. For example, statement 1 succeeds the entry of the method and can be succeeded by statement 2 or 4 at runtime. Labeled edges represent branches or control decisions (e.g., T for true).

Figure 1(c) shows the Program Dependence Graph (PDG) [12, 13] for the example program `E`. The graph has a special node START. Edges T from this node represent the decision to execute the program. A solid edge in the PDG denotes the *control dependence* [12] of the target node on the decision taken at the source

[1] Available at: `http://nd.edu/~rsanteli/duaf`

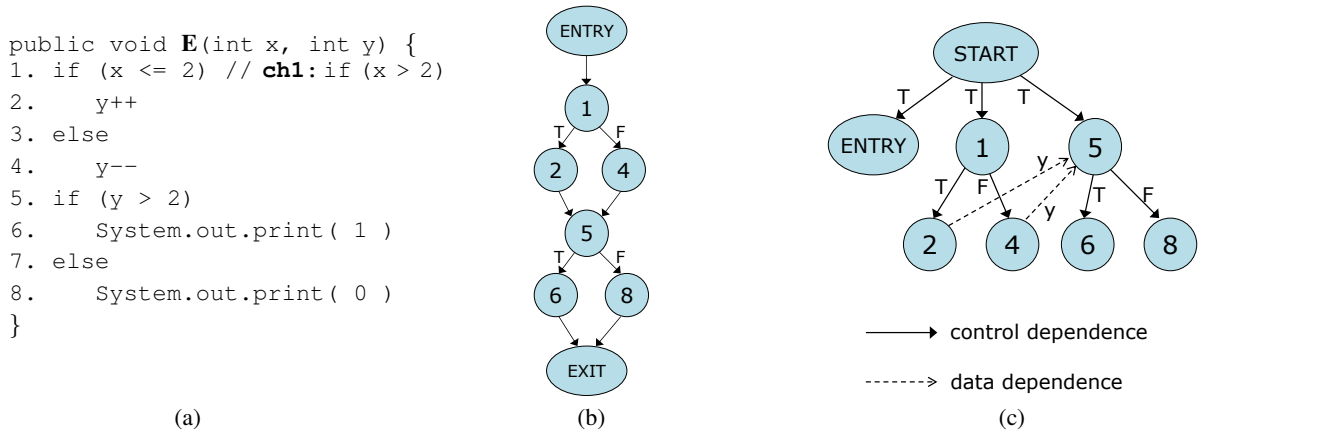

Figure 1. Simple example method `E` with a change `ch1` (a), its control-flow graph (CFG) (b), and its program dependence graph (PDG) (c).

node. For example, in program `E`, statement 2 is control dependent on statement 1 evaluating to *true*. A dashed edge denotes a *data dependence* [2] of the target node on the source node. For example, statement 5 is data dependent on statement 2 for variable `y` because 2 defines `y` and that definition reaches the use of `y` in statement 5.

3. Architecture of DUA-FORENSICS

DUA-FORENSICS normally works in three phases. Figure 2 illustrates this process. The phases are, in order:

1. The *static*-analysis phase, which inputs the subject program and analyzes this program to identify information such as control and data dependencies. Typically, this phase also instruments the subject to track this information at runtime.
2. The *runtime* phase takes this information and the instrumented subject and, as the user runs the instrumented subject, collects *dynamic* information that the instrumentation produces.
3. A *post-processing* phase, which is needed for non-trivial analyses to obtain the desired results. This phase removes much of the overhead burden from the runtime phase and makes it easier to put together the results of multiple executions later.

Figure 3 shows the *internal* architecture of the static phase, which is the most important, and which relies heavily on Soot. This diagram shows the instrumentation modules on the top left. An abstraction called *probe* on the middle left adds a layer on top of Soot to facilitate instrumentation for any purpose and for any user. In DUA-FORENSICS, we exploit this abstraction to enhance Soot's API and implement three core instrumenters: branches, data dependencies, and dependence chains (see Sections 4 and 6).

The instrumenters determine *what* to instrument by asking the fine-grained dependence analysis module, which is shown on the top right in Figure 3. This module takes advantage of the library-call modeling feature (middle right of the figure) to skip the analysis of libraries by Soot and DUA-FORENSICS and, thus, speed up the overall dependence analysis (at the potential expense of some accuracy). We discuss these modules in the next three sections.

4. Instrumentation Layer

4.1 Probe Abstraction

DUA-FORENSICS expands Soot's Jimple instrumentation capabilities with a higher-level feature called *probe* and new instrumentation algorithms. A probe encapsulates all instructions that users want to insert in a `UnitGraph`'s (CFG) entry, edge, or node (before the node's `Stmt` executes). The layer ensures that all of the inserted code in the probe executes at that point without breaking the control-flow graph of the program. The layer redirects all `UnitGraph` edges that target a node to the beginning of the probe for that node, so that the probe executes before the node.

Using probes, users can apply multiple types of instrumentation while preserving, after each instrumentation pass, the distinction between the original and the inserted code to allow subsequent passes. Users can choose to insert new code at the top or bottom of a probe to control the order in which events are reported at runtime. We have used these abilities to mix branch, data dependence, and dependence-chain instrumentation without conflicts [25, 26].

4.2 Branch Instrumenter

For branch (edge) coverage, the instrumenter in DUA-FORENSICS offers two options to the user: instrument all branches directly or the (theoretically) optimal branch-instrumentation algorithm of Ball and Larus [7]. In practice, we found the runtime overhead of both approaches to be the same in our benchmarks [22]. We have not yet investigated the reasons for this phenomenon.

For efficiency at runtime, we used an array of bytes to represent branch coverage. When a branch executes, the initial value of 0 for that branch changes to 1. At the end of the subject's execution, the instrumentation reports the values in this array. To avoid missing this report when the program ends early due to unhandled exceptions, our instrumentation wraps the entry method in a try-catch block that redirects execution to this reporting code.

4.3 Data-Dependence Instrumenter

For data-dependence coverage, DUA-FORENSICS uses the *last-definitions* approach [19, 22]. This approach maintains a table at runtime that maps to each variable that has not been garbage-collected the location of its last definition. When a use is reached, the runtime monitor looks up the last definition for that variable—a `Local`, a `SootField`, or a library-object abstraction (see Section 5)—and reports the coverage of the corresponding data dependence. We use `WeakReference`s to refer to those variables at runtime to avoid interference with the garbage collector.

4.4 Discussion

All these instrumentation enhancements to Soot—probes, branch instrumentation, and data-dependence instrumentation—should be of interest to Java researchers and developers. These are also the building blocks we used to develop more complex instrumentation types, such as dependence chains, which we defer to Section 6.

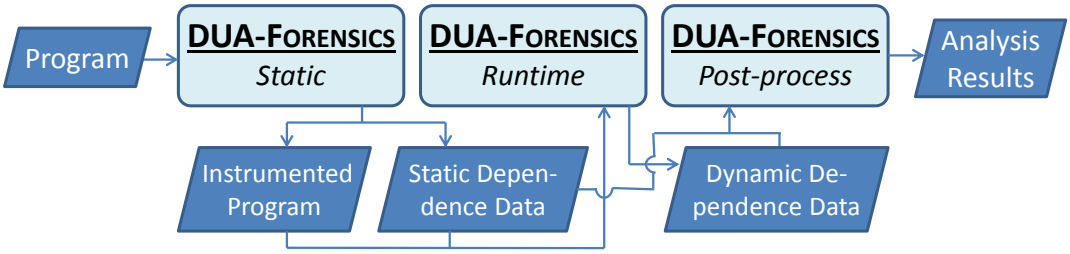

Figure 2. Architecture of DUA-FORENSICS showing its three phases: static analysis and instrumentation, runtime, and post-processing.

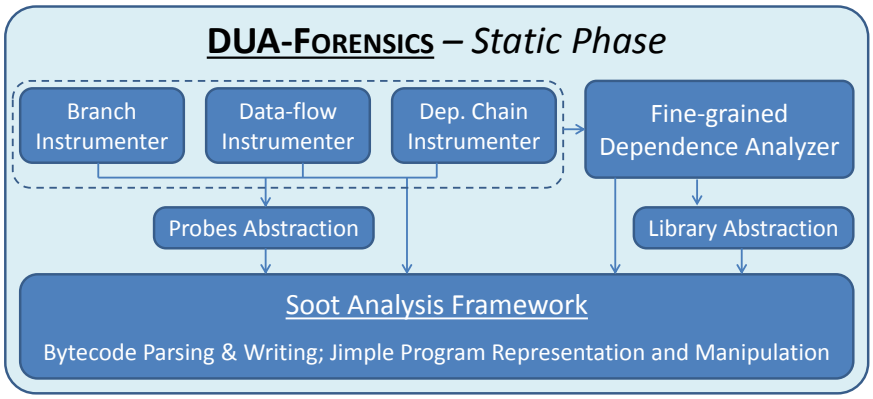

Figure 3. Internal architecture of the static phase of DUA-FORENSICS, highlighting its main components and Soot.

5. Library-Call Modeling

In our research, we focus on the behavior of the subject program rather than the libraries it may (transitively) call. To avoid the long processing times often required for whole-program analysis in Soot, while still capturing the effects of library calls, we added to DUA-FORENSICS method-call abstractions (summaries) for points-to-analysis, data-flow analysis, and symbolic execution. These abstractions let users model instances of library classes as special kinds of variables and encode the effects of method calls on those objects and on regular Soot variables.

5.1 Points-to Analysis

For points-to analysis, instead of modifying the existing algorithms in Soot, we decided to re-implement Andersen's context- and flow-insensitive algorithm [3] in a way that merges seamlessly with our new library-call points-to effects abstraction. For this abstraction, we defined the base class `AbstractP2Model`:

```
public abstract class AbstractP2Model {
  protected static List<Pair<Value,Value>> empty;
  public List<Pair<Value,Value>>
   getSeeds(InvokeExpr invExp) {return empty;}
  public List<Pair<Value,Value>>
   getTransfers(InvokeExpr invExp) {return empty;}
}
```

For an `InvokeExpr` that calls (or may call) a library method, this class defines, as default behavior, that no *seeds* (object creations) and no *transfers* (assignments of points-to sets) occur. For subclasses of `AbstractP2Model`, each pair of Soot `Values` returned indicates that the points-to set of the first `Value` must be updated with the points-to set of the second `Value`. For `getSeeds`, the points-to set of the second variable is a new abstract memory location. If the first value is `null`, the points-to set of the second value is added to the points-to set of the left-hand-side variable in the containing `Stmt` (if it is an assignment). Our subclasses of `AbstractP2Model` cover the behavior of most library calls we have found. Users can also add their own subclasses.

An important feature of this system is the ability to define *abstract fields*, such as the field of a container (e.g., a `List`) that represents all elements in it. For other containers, such as `Map`, we use two abstract fields—one to represent keys and another to represent values. Although we do not analyze the contents of libraries—and these fields most likely do not exist in their implementations exactly as we model them—these constructs let us propagate points-to sets among real variables in the program through library-object models that the program uses as intermediaries.

Our points-to analysis is completed by a table that maps fully-qualified method signatures to the model classes that match their behavior. We have filled this table manually with the most common library calls we have found. A current list of supported library calls can be found in class `dua.global.p2.P2ModelManager`.[1] In the near future, we plan to add automatic support to fill this table to avoid the costs and risks of manual modeling.

5.2 Dependencies and Symbolic Execution

For dependence analysis and symbolic execution, we model the effects of library calls in a similar way. For data-dependence analysis, overridable methods for library-call models are offered which, given a Soot `InvokeExpr`, return the variables—`Values` or abstract library objects—definitely or possibly defined and used during the call. For symbolic execution, the model returns (possibly conditional) symbolic expressions for updated variables based on the `InvokeExpr` and the path condition for reaching that call.

5.3 Discussion

The trade-offs between precision and speed of using library models still need to be formally studied. However, we have seen strong indications that these trade-offs, at least for our fine-grained dependence-based research tasks, are quite useful—our research techniques have proven scalable. For instance, for XML-security [4], one of our typical subjects which has about 20 KLOC, our points-to analysis takes just a few seconds, whereas Soot's default analysis takes several minutes on a modern machine.

6. Fine-Grained Dependence Analyzer

Soot and our instrumentation and library-modeling features are the foundations for our fine-grained dependence analyzer. DUA-FORENSICS started as a DUA (Definition-Use Association) monitoring and inferencing tool that speeds up data-flow coverage and approximates it from branch coverage for testing [22]. This inferencing ability is the reason for our tool's name. We also added to this component a fault-localization module that simultaneously exploits coverage information for statements, branches, and data dependencies (precise or inferred) [28]. Later, we extended DUA-FORENSICS to analyze and monitor the *chains* (sequences) of data and control dependencies that propagate the effects of changes in software [25, 27, 29] and to quantify those effects [24, 30, 34].

In 2006, Soot did not yet provide control-dependence analysis. Therefore, one of our first tasks in DUA-FORENSICS was to implement an intra- and inter-procedural control-dependence analysis for Soot. For dependencies created by virtual calls, we used class-hierarchy analysis [10]. In addition, we extended the existing intra-procedural analyses to identify inter-procedural data dependencies. Then, using these dependencies, we implemented reachability analyses in our inter-procedural dependence graph. As our goals have been to *statically* analyze short sequences of dependencies [23, 27] and to *dynamically* analyze longer sequences [25, 26, 29], we have not yet made our static reachability algorithms context-sensitive. For such sequences, there is little or no loss of precision.

Given the ability to perform transitive closures on the static and dynamic dependence graphs, the users of DUA-FORENSICS can perform (context-insensitive) static slicing and dynamic slicing. Of all granularities of dynamic slicing [1], DUA-FORENSICS uses the finest-grained version which distinguishes among occurrences of dependencies. Also, our tool identifies which occurrences connect with which other occurrences. For our research, this is crucial as we can obtain not only dynamic slices, but also dynamic dependence paths—*chains* (sequences) of dependencies which correspond to the paths that can propagate the effects of errors and changes.

For a simple example, consider the PDG of the example program `E` in Figure 1. A fine-grained forward analysis of the potential effects of change `ch1` in statement 1 shows that there are four dependence chains along which the change can propagate to the output statements 6 and 8:

```
⟨(1,2), (2,5), (5,6)⟩,
⟨(1,2), (2,5), (5,8)⟩,
⟨(1,4), (2,5), (5,6)⟩, and
⟨(1,4), (2,5), (5,8)⟩.
```

This is, however, a trivial example. In general, the number of chains can grow exponentially with the size of the program. For programs with loops or recursion, that number can be infinite. Hence, we allow length and iteration limits when analyzing these chains.

To determine the coverage of chains, the chain instrumentation is activated every time the change point (e.g., statement 1 in `E`) is reached. At that point, the occurrence of this node is marked as an *open* source-node occurrence from which one or more dependencies might be covered. When covered, each such dependence is recorded and the occurrence of its target node is marked as open. Because tracking all links among dependence occurrences at runtime is taxing, we only record coverage and opening events. Later, at post-processing (Figure 2), we determine the covered chains.

Finally, DUA-FORENSICS allows to collect the values computed at each dependence node. This is achieved by identifying, for each node, the variables (real or abstract) defined by a node and logging their values along with each dependence event. This information is necessary to determine whether there is a *semantic dependence* [21, 29] of a node on a change at runtime. A node in a chain depends semantically on a change if its execution history or values change. If that dependence is observed, then the change has propagated along that chain to that node.

7. Application: Change Testing

Testing a program change involves obtaining a picture as complete as possible of how the change affects the behavior of the program. This is akin to shining a light on an object (the change) and looking at the projected shadows. To that end, with the help of Soot, DUA-FORENSICS implements MATRIX, a set of static and dynamic analyses for identifying (abstractions of) the effects of a change and for exercising those effects [5, 25–27].

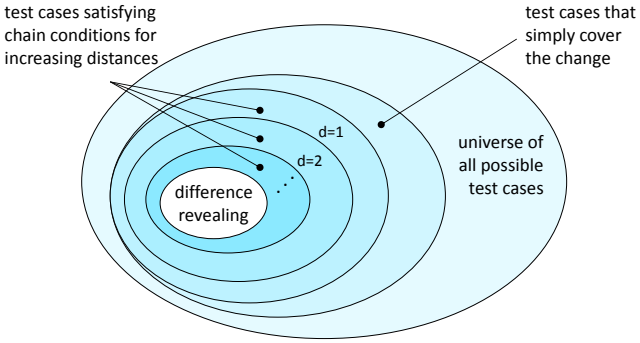

Figure 4. Intuitive view of how MATRIX increasingly approximates the ideal solution with each distance.

7.1 The MATRIX Technique

Figure 4 gives an intuitive view of how MATRIX works. MATRIX computes testing requirements (conditions) for the subset of tests that change some state or path in the program after it is modified. The goal is to make this change *observable* at the output. This subset is first approximated by the tests that cover the change and then is increasingly better approximated by the tests that cover propagating dependence chains of increasing distances from the change. The greater the distance is, the closer the tests get to the output-difference revealing subset (but the greater the cost is).

Our latest approach to MATRIX is dynamic and demand driven [25, 26]. Instead of computing all test requirements beforehand, DUA-FORENSICS identifies the unsatisfied requirements at the *frontier* of the requirements already satisfied. Given a test suite *TS*, a *frontier test requirement* is a dependence chain that is covered by *TS* up to, but excluding, its last dependence. A frontier test requirement for *TS* includes the propagation conditions along the corresponding chain. Consequently, the set of unsatisfied test requirements for test suite *TS* is the union of the frontier test requirements for *TS* and every other test requirement whose chain extends the chain of a frontier test requirement.

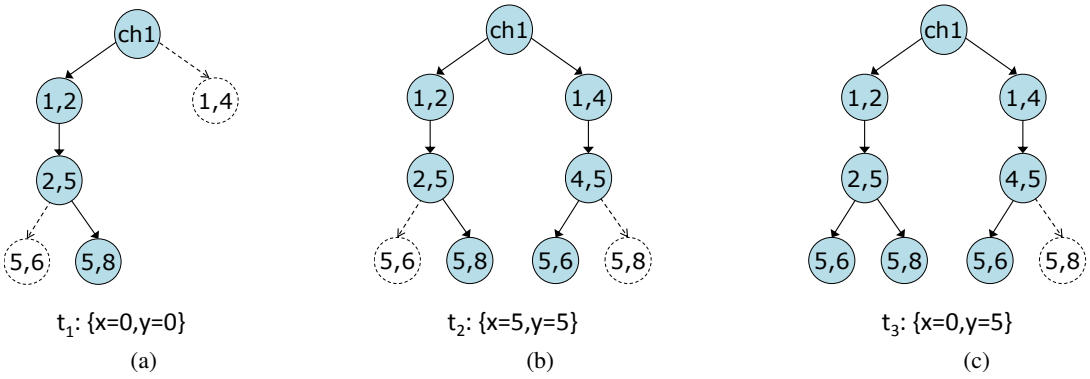

Figure 5. Chain trees for satisfied and frontier test requirements (shaded solid and unshaded dashed circles, respectively) for program `E` of Figure 1. In steps (a), (b), and (c), the new test cases t_1, t_2, and t_3 incrementally expand the chain tree for that program and its test suite.

To illustrate, we use *chain trees* to denote satisfied and frontier test requirements for changes. The root node of a change tree represents the change and each other node pairs a dependence and its propagation condition. Each such pair is either part of a satisfied test requirement or the end of a frontier test requirement. A node at the end of a frontier test requirement is called *frontier node* and is a leaf in the chain tree. The root node is annotated with an identifier for the change and every other node is annotated with the corresponding dependence. Each edge (u, v) in a chain tree indicates that the dependence and propagation condition for node u (or the change, if u is the root) are followed by the dependence and propagation condition for node v in some test requirement. Therefore, the target statement of the dependence for tree node u is the source statement of the dependence for tree node v.

Consider program `E` and change `ch1` from Figure 1. Before any test cases execute, the chain tree for `ch1` in `E` consists of only one node—the frontier node for change `ch1` in statement 1. Figure 5 shows this chain tree growing as three different test cases are added, one after another. The first tree results from adding test case t_1, which traverses chain $\langle$(1,2), (2,5), (5,8)$\rangle$ in `E`. The nodes belonging to this satisfied test requirement are shown as shaded solid circles. The frontier nodes, for dependencies (1,4) and (5,6), are shown as unshaded dashed circles. These dependencies, which succeed the change `ch1` and dependence (2,5), respectively, are identified by DUA-FORENSICS as the next targets—any chain covered next will have chain $\langle$(1,4)$\rangle$ or $\langle$(1,2), (2,5), (5,6)$\rangle$ as a prefix.

In the middle and on the right, Figure 5 shows the results of adding two more test cases. Test case t_2 expands the chain tree via frontier node (1,4) by covering chain $\langle$(1,4), (4,5), (5,6)$\rangle$. Because this chain includes dependence (1,4), (1,4) is no longer a frontier node. Instead, node (5,8) becomes a frontier node that succeeds (4,5) in the tree because, after covering dependence (4,5), t_2 continues to its other successor—dependence (5,6). Then, test case t_3 covers dependence (5,6) after $\langle$(1,2), (2,5)$\rangle$. Thus, (5,6) on the bottom left is no longer a frontier node after adding t_3.

7.2 Empirical Evaluation

We evaluated this demand-driven approach for change testing using DUA-FORENSICS. Our goal was to determine whether, and by how much, MATRIX increases the chances of success when testing a change with respect to RAND (just covering the change), BR (covering affected branches), and DU (covering affected du-pairs).

7.2.1 Subjects and Methodology

Our subjects are listed in the first column of Table 1. We chose Java programs for which a large number of test cases are available so we could simulate the creation of tests under various techniques. The first five subjects are from the Siemens suite [14], which we translated from C to Java. We also obtained three releases of NanoXML from the SIR repository [11]. For each subject, we obtained between six and nine changes, for a total of 62 changes.

Table 1. Average ratios at which the studied change-testing techniques revealed output differences per added test.

subject	RAND	BR	DU	CHAIN$_d$	PROP$_d$
Tot_info	0.31	0.23	0.27	0.35	**0.51**
Schedule1	0.06	0.09	0.08	0.10	**0.30**
Schedule2	0.23	0.19	0.15	0.17	**0.33**
Print_tokens1	0.19	0.30	0.29	0.48	**0.57**
Print_tokens2	0.38	0.49	0.47	0.53	**0.57**
NanoXML v1	0.45	0.37	0.42	0.45	**0.54**
NanoXML v2	0.53	**0.56**	0.53	0.55	0.55
NanoXML v3	0.53	0.37	0.55	0.59	**0.69**

For each change in each subject and for each technique, we performed multiple test-suite augmentations by adding tests from the pool, including two versions. For MATRIX, we included all distances d from 1 to 10 and studied two variants: CHAIN$_d$ (just chain coverage, without propagation conditions) and PROP$_d$ (the full technique). After performing all augmentations, for each technique, we divided the number of tests that succeeded at causing output differences by the total number of tests added. Each result is a number in [0,1] indicating the ratio of success.

The runtime overhead of using MATRIX was about 600% on average, which is not much higher than the typical 100%-200% overhead incurred by data-dependence monitoring.

7.2.2 Results and Analysis

Table 1 shows the ratios at which each technique found output differences per added test for each subject. For CHAIN$_d$ and PROP$_d$, we report the average ratios for distances 1 to 10. The best result per subject is highlighted in bold.

For seven out of eight subjects, PROP$_d$ was the best technique. For the other subject, PROP$_d$ was almost as good as the best. A Wilcoxon signed-rank test [33] suggests that the superiority of MATRIX (especially PROP$_d$) over the other approaches is significant with 99.9% confidence. Also, CHAIN$_d$ seems an acceptable alternative to PROP$_d$ that avoids the cost of collecting runtime values.

In all, the variety of features that DUA-FORENSICS offers, including fine-grained dependence and propagation analysis, allowed us to validate this novel and detailed technique for testing changes.

8. Conclusion and Future Directions

We presented DUA-FORENSICS, a system that exploits core features of Soot to offer detailed instrumentation, modeling, and analysis of dependencies in Java programs. This tool has supported our research by assuring the reliability of our technique implementations. DUA-FORENSICS is open source and available online.[1]

The technical challenges of integrating DUA-FORENSICS into Soot depend on each feature. The instrumentation and dependence-analysis layers are built on top of Soot, which facilitates this task. The library-call models can also be fully integrated as soon as the existing analyses in Soot support these new kinds of variables.

In the future, more work is needed to better exploit all features of Soot in DUA-FORENSICS, such as object-sensitive points-to analysis [18] and the new inter-procedural framework [8]. We expect to remove the redundancies that exist with the latest versions of Soot and offer our system as a proper extension of Soot.

We expect that DUA-FORENSICS will continue to improve in performance and features to cater to the analysis community and to support our ongoing efforts to quantify the effects of changes. DUA-FORENSICS could become the reference implementation for dependence- and change-analysis applications. We also intend to study and connect with WALA [15], Chord [20], and Indus [16], which share common goals with Soot and DUA-FORENSICS.

Acknowledgments

We thank Mary Jean Harrold for her support over many years, which allowed us to create DUA-FORENSICS. We also thank the Soot team for creating and supporting the Soot framework.

References

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proc. of ACM Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools (2nd Ed.)*. Sept. 2006.

[3] L. O. Andersen. Program analysis and specialization for the C programming language. *Ph.D. Thesis, DIKU, U. of Copenhagen*, May 1994.

[4] Apache Santuario project. Apache XML Security for Java. Apache Software Foundation. `http://santuario.apache.org/`.

[5] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proc. of TAIC-PART*, pages 137–146, Aug. 2006.

[6] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. Georgia Institute of Technology. `http://pleuma.cc.gatech.edu/aristotle/Tools/jaba.html`.

[7] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[8] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proc. of ACM SIGPLAN Int'l Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 3–8, June 2012.

[9] M. Dahm. Byte Code Engineering. In *In Java-Informations-Tage*, pages 267–277, Sept. 1999.

[10] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of ECOOP*, pages 77–101, Aug. 1995.

[11] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Emp. S. Eng.*, 10(4):405–435, 2005.

[12] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319-349, July 1987.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Prog. Lang. and Systems*, 12(1):26-60, Jan. 1990.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of Int'l Conf. on Softw. Eng.*, pages 191–200, May 1994.

[15] IBM T.J. Watson Research Center. T.J. Watson Libraries for Analysis (WALA). IBM. `http://wala.sourceforge.net`.

[16] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *In Proc. of FASE*, pages 269–272, Apr. 2005.

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

[18] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proc. of Int'l Conf. on Compiler Construction*, pages 47–64, Mar. 2006.

[19] J. Misurda, J. Clause, J. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proc. of Int'l Conf. on Softw. Eng.*, pages 156–165, May 2005.

[20] M. Naik. Chord: A Versatile Platform for Program Analysis. In *Tutorial at ACM Conference on Programming Language Design and Implementation*, June 2011.

[21] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[22] R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *Proc. of Int'l Conf. on Automated Softw. Eng.*, pages 343–352, Nov. 2007.

[23] R. Santelices and M. J. Harrold. Exploiting Program Dependencies for Scalable Multiple-path Symbolic Execution. In *Proc. of ACM Int'l Symp. on Softw. Testing and Analysis*, pages 195–206, July 2010.

[24] R. Santelices and M. J. Harrold. Probabilistic slicing for predictive impact analysis. *Tech. Rep. CERCS-10-10*, Georgia Tech, Nov. 2010.

[25] R. Santelices and M. J. Harrold. Applying Aggressive Propagation-based Strategies for Testing Changes. In *Proceedings of Int'l Conference on Softw. Testing, Verif. and Valid.*, pages 11–20, Mar. 2011.

[26] R. Santelices and M. J. Harrold. Demand-driven Propagation-based Strategies for Testing Changes. *J. of Software Testing, Verification and Reliability*, 2013. To appear.

[27] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. of Int'l Conf. on Autom. Softw. Eng.*, pages 218–227, Sept. 2008.

[28] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight Fault Localization Using Multiple Coverage Types. In *Proc. of Int'l Conf. on Softw. Eng.*, pages 56–66, May 2009.

[29] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proc. of Third IEEE Int'l Conf. on Softw. Testing, Verif. and Valid.*, pages 429–438, Apr. 2010.

[30] R. Santelices, Y. Zhang, S. Jiang, H. Cai, and Y. jie Zhang. Quantitative Program Slicing: Separating Statements by Relevance. In *Proc. of Int'l Conf. on Softw. Eng., New Ideas and Emerging Results*, pages 1269–1272, May 2013.

[31] A. Seesing and A. Orso. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proc. of Eclipse Technology eXchange (eTX) Workshop at OOPSLA*, pages 49–53, Oct. 2005.

[32] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research*, pages 13–23, Nov. 1999.

[33] R. E. Walpole, R. H. Myers, S. L. Myers, and K. E. Ye. *Probability and Statistics for Engineers and Scientists (9th Edition)*. Prentice Hall, Jan. 2011. ISBN 978-0321629111.

[34] Y. Zhang and R. Santelices. Predicting Data Dependences for Slice Inspection Prioritization. In *Proc. of IEEE Int'l Workshop on Program Debugging*, pages 177–182, Nov. 2012.