Recovering Execution Data from Incomplete Observations

Peter Ohmann* David Bingham Brown* Ben Liblit* Thomas Reps*†

*University of Wisconsin–Madison, USA †GrammaTech, Inc., USA

{ohmann, bingham, liblit, reps}@cs.wisc.edu

Abstract

Due to resource constraints, tracing production applications often results in incomplete data. Nevertheless, developers ideally want answers to queries about the program's execution beyond data explicitly gathered. For example, a developer may ask whether a particular program statement executed during the run corresponding to a given failure report.

In this work, we investigate the problem of determining whether each statement in a program executed, did not execute, or may have executed, given a set of (possibly-incomplete) observations. Using two distinct formalisms, we propose two solutions to this problem. The first formulation represents observations as regular languages, and computes intersections over these languages using finite-state acceptors. The second formulation encodes the problem as a set of Boolean constraints, and uses answer set programming to solve the constraints.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, testing tools, tracing; D.2.8 [*Software Engineering*]: Metrics—Performance measures

General Terms Algorithms, Languages, Theory

Keywords Debugging, program coverage, execution tracing

1. Introduction

Much prior research exists on gathering various levels of program coverage [1, 5, 6, 22]. In practice, however, execution observations are often incomplete, especially for deployed applications. For example, to reduce run-time overheads, an instrumented program might randomly sample observation points [14] or trace only a small portion of execution [16]. Sampling-based profilers observe a program's state at regular time intervals, but do not observe a static set of logging points. In general, many different runs could produce the same incomplete observation data.

In this work, we address what we call the *incomplete-observations problem*. The goal is to extract statement-coverage information from possibly incomplete observation data. Incomplete observations may result in incomplete coverage information; our task is to reconstruct as much information as possible given these limited observations. We formalize the incomplete-observations problem in a way that applies both to instrumentation-based and sampling-based tools. The input is a *failure report*, which consists of:

1. a program P with control-flow graph (CFG) G
2. a node $crash \in G.\text{nodes}$ marking the end of execution (i.e., the crash location)
3. a set $obsYes \subseteq G.\text{nodes}^+$ whose elements are non-empty sequences over G.nodes. Each set element corresponds to a sequence of statements that are observed to have executed in the given order. Ordering between elements of distinct *obsYes* sequences is unspecified.
4. a set $anyObsYes \subseteq G.\text{nodes}$ defined as the set of program locations appearing in any *obsYes* sequence
5. a set $obsNo \subseteq G.\text{nodes}$ representing a set of program locations that are known to not have executed.

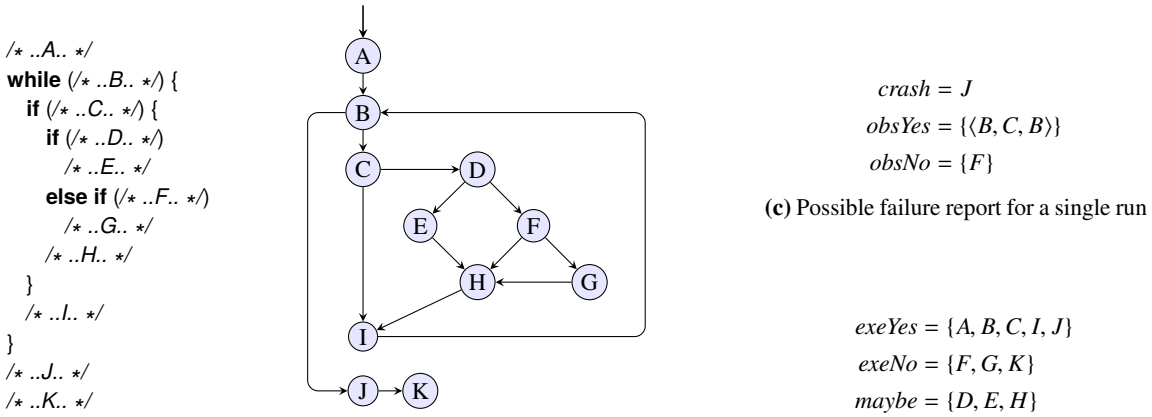
Any node in G might appear in multiple *obsYes* sequences, or more than once in any single *obsYes* sequence.

Consider two example cases. First, the csi-cc tool of Ohmann and Liblit [16] gathers basic statement-coverage data at a limited set of program locations. The information gathered corresponds to the degenerate case in which each *obsYes* sequence consists of one element: each observation is independent, and ordering among observations is unspecified. All instrumentation points that are not in *anyObsYes* are in *obsNo*. Second, sampling-based profilers always yield an *obsYes* consisting of a single sequence corresponding to an incomplete trace. In this case, *obsNo* is always empty, because the profiler does not provide information about whether a statement *never* executes. The above formulation cannot express the common case of reliable logging (e.g., with `printf` statements). See Section 5 for further discussion.

anyObsYes and *obsNo* are disjoint, but may not form a partition of G.nodes. Typically, most CFG nodes are not

WODA'15, October 26, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2823363.2823368



(a) Code skeleton **(b)** Control-flow graph **(c)** Possible failure report for a single run **(d)** Corresponding desired analysis result

Figure 1. An example program with crash data

directly observed at all; our challenge is to infer the behavior of unobserved nodes based on the few that we did observe.

Specifically, an answer to the incomplete-observations problem is a partition of G.nodes into subsets *exeYes*, *exeNo*, and *maybe*. *exeYes* contains those nodes in G that necessarily exist on all paths through G that are consistent with the failure report. Conversely, *exeNo* contains those nodes that cannot exist on any such path, while *maybe* contains nodes that exist on some but not all paths consistent with the failure report. A path r is consistent with a failure report if (i) $r_{|r|} = crash$, (ii) no node from *obsNo* occurs in r, and (iii) every *obsYes* sequence is a subsequence of r. Suppose that f is a failure report and $R = \{\text{paths through } G \text{ consistent with } f\}$. Then,

$$
\begin{aligned}
exeYes &= \{m \mid \forall\, r \in R, m \text{ occurs in } r\} \\
exeNo &= \{m \mid \forall\, r \in R, m \text{ does not occur in } r\} \\
maybe &= G.\text{nodes} - exeYes - exeNo
\end{aligned}
$$

Figure 1a shows an example code skeleton, and Figure 1b shows the corresponding CFG. Figure 1c gives partial trace information from an example (crashing) run of the program; the program crashes at statement J after observing the execution of statement B, followed by the execution of statement C, followed by another execution of statement B. Statement F is a member of *obsNo*; thus, F did not execute on this run. Figure 1d shows the ideal *exeYes*, *exeNo*, and *maybe* sets, given the failure report in Figure 1c.

This problem is interesting for a variety of reasons. Testing and debugging accounts for 50–75% of a software project's cost [9, 12, 21]. When debugging from failure reports, developers ask questions about the application's execution, and a developer may want to know which parts of his/her program executed, and which parts did not execute. In a more foundational sense, the incomplete-observations problem is an example of partially-dynamic program analysis, and involves extracting optimal information while reasoning in the presence of uncertainty. We discuss this topic further in Section 4.

2. Two Formulations

In this section, we describe two distinct approaches to solve the incomplete-observations problem[1]. In both cases, our approach separates the encoding of the program (in this case, the CFG) from the encoding of observations specific to a given run of the program. The analysis need only encode the program's CFG once. We encode any failure data (including the crashing location) as additional constraints over paths in the CFG, as represented in the appropriate formalism.

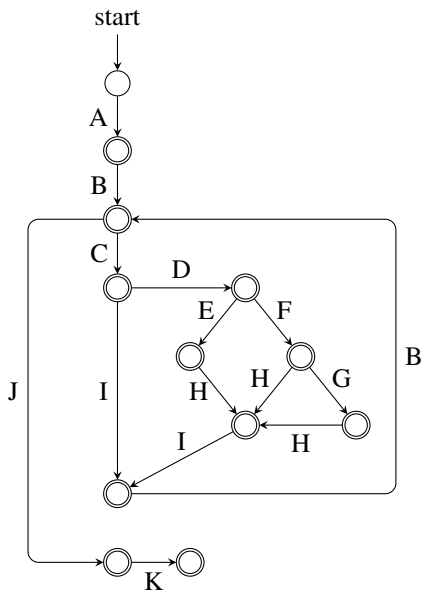
Our first formulation encodes a failure report using standard finite-state automata (FSA). In this approach, we express paths in the CFG as strings in the language accepted by the FSA. Our second formulation is based on answer set programming (ASP) [15]. In this approach, we encode the execution of each statement as a predicate in a logic program, and express a failure report as constraints over these predicates. We believe that the formulations are equivalent. Section 3 provides evidence for (but no proof of) this claim.

2.1 Automata

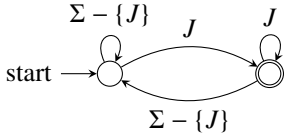
A deterministic finite-state acceptor (FSA) is defined as $A = (Q, q_0, F, \Sigma, \delta)$ where:

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final (i.e., accepting) states,
- Σ is a finite alphabet of symbols, and
- $\delta : Q \times \Sigma \to Q$ is the deterministic transition function

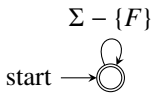
[1] Complete formulations and additional details are available at `http://pages.cs.wisc.edu/~liblit/woda-2015/extended/`.



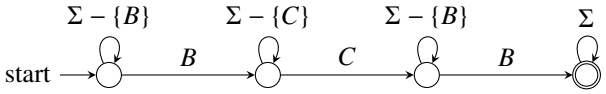
(a) A_G, the control-flow graph encoding



(b) *crash* encoding



(c) *obsNo* encoding



(d) *obsYes* encoding

Figure 2. Finite-state acceptor formulation of the example from Figure 1

The language, $L(A)$, recognized by A consists of all strings s in Σ^* such that there exists a corresponding sequence of states v in Q^* where $v_0 = q_0$, $v_{|s|} \in F$, and $\forall\, i \in [0, |s| - 1]$. $v_{i+1} = \delta(v_i, s_i)$. In other words, A accepts a string in Σ^* if it corresponds to legal transitions through A ending in an accepting state.

For a program P with CFG G, we encode executions through P as strings in a language L_G whose alphabet Σ is G.nodes. Encoding G as an FSA A_G is straightforward. A_G's states are the nodes of G with a new initial state q_0; that is, $Q = G.\text{nodes} \cup \{q_0\}$. The transitions of A_G correspond to the edges of G: each (x, y) in G.edges contributes $q_x \xrightarrow{y} q_y$ to δ. In addition, if e is the entry node of G, δ contains a transition $q_0 \xrightarrow{e} q_e$. All states of A_G except q_0 are accepting states; that is, $F = Q - \{q_0\}$. This automaton is not specific to the location of the crash site; it accepts any prefix of a valid execution, no matter where the partial execution stops. Figure 2a shows the FSA corresponding to the CFG in Figure 1b.

As defined above, $L(A_G)$ (the language accepted by A_G) reflects all possible paths in G, without regard to *crash*, *obsNo*, or *obsYes*. To incorporate these items, we create additional FSAs: one for *crash*, one for *obsNo*, and one for each entry in *obsYes*. These FSAs serve to further constrain the set of possible paths. Let C be the set of additional FSAs representing these constraints, and let A_G^C be the intersection of A_G with all $c \in C$. Then A_G^C is the automaton that recognizes only the paths through G that are consistent with the given failure report.

We derive these additional constraint FSAs from a failure report as follows. To encode the crash site, we create an FSA that accepts any input string ending in *crash*. Figure 2b shows this FSA for our example. The *obsNo* constraint merely asserts that no element of *obsNo* appears in an accepted string; Figure 2c encodes this constraint. Constraint FSAs for *obsYes* entries encode ordered statement observations and allow any number of transitions on Σ after the final observed statement. Figure 2d encodes the $\langle B, C, B\rangle$ constraint from the example in Figure 1. Note that the automata from Figures 2b–2d allow the crashing CFG node, J, to occur more than once during an execution. This behavior is impossible for the CFG from Figure 2a, but is possible (and even necessary) in general.

To compute *exeYes*, *exeNo*, and *maybe*, we then iterate through each $n \in G$.nodes and construct two n-specific constraint automata that serve as probes for A_G^C. Let $obsYes_n$ be the *obsYes* automaton that corresponds to the single-element sequence $\langle n\rangle$. Let $obsNo_n$ be the *obsNo* automaton corresponding to the singleton set $\{n\}$. If $L(A_G^C \cap obsYes_n)$ is empty, then $n \in exeNo$. If $L(A_G^C \cap obsNo_n)$ is empty, then $n \in exeYes$. Otherwise, $n \in maybe$.

Intersecting the constraint automata (specifically the *obsYes* automata) could result in an exponential increase in the number of transitions in A_G^C. Our evaluation (Section 3) indicates that this can be a problem in practice.

As described, this approach applies to intraprocedural analysis. However, our formulation also extends to interprocedural analysis. In this case, G is the interprocedural CFG of P, meaning that call statements have edges to the corresponding function entry, and return statements in the callee have edges to all corresponding return locations in the caller. If we directly apply our intraprocedural approach to the in-

terprocedural CFG (with our initial state preceding the entry to the main function), we obtain a context-insensitive result. That is, this result does not enforce, for each matched path r through G, that return statements in r are correctly matched with corresponding call statements earlier in r. Obtaining a context-sensitive result requires that we move to a more powerful automaton formulation, such as visibly pushdown automata or nested word automata [3]. As noted by Alur and Madhusudan, both structures can be checked for emptiness and are closed under intersection. Future evaluation will determine if more expressive automata increase analysis costs.

2.2 Answer Set Programming (ASP)

There has been an enormous amount of work in recent years on using Boolean satisfiability (SAT) solvers in program analysis and model checking. A SAT solver attempts to find a satisfying assignment to a set of Boolean variables given constraints over those variables. For a CFG G, we would like to encode information about a path r (consistent with our failure report) with one Boolean variable for each $n \in G$.nodes to represent whether n occurs in r. Unfortunately, our problem requires that we reason about graph connectivity; thus, our solution cannot be expressed in first-order logic.

Instead, we must use a recursive language for constraints, and chose the logic programming paradigm. Fortunately, our problem has a finite number of atoms; we can use ASP to convert our logic program to a Boolean ground theory in conjunctive normal form. This process, *grounding*, serves as an interface to a SAT solver. The syntax used in our implementation (the Gringo [10] grounder's native format) is inspired by other logic-programming languages, such as Datalog, though its evaluation differs significantly. Rather than directly evaluating a logic program, an ASP system first converts the program into SAT form, allowing it to benefit from existing SAT technology.

Our logic program separates the encoding of G's structure, the constraints from items 2–5 of a failure report, and the general constraints for an incomplete-observations problem. The node, edge, obsNo, anyObsYes, and crash predicates are derived directly from a failure report. The order predicate encodes pairwise orderings expressed by *obsYes* sequences. For example, the $\langle B, C, B\rangle$ constraint from Figure 1 would require two facts: "order(b, c)." and "order(c, b)."

Several program-independent background constraints support this simple encoding scheme, as shown in Figure 3. In the end, each satisfying model for our constraints describes a set of paths through G consistent with a specific failure report. Note that any cyclic CFG has an infinite number of paths. However, a model represents a family of paths as defined by visited and connected. In the worst case, the number of such models is exponential; we revisit this below.

Line 1 in Figure 3 states that the remaining definitions of visited specify a partial (rather than complete) set of truth values. In other words, the rules in lines 3–6 may not entirely determine the status of some nodes. These rules

```
1  0 { visited(X) } 1 :- node(X).
2
3  visited(X) :- entry(X).
4  visited(X) :- crash(X).
5  -visited(X) :- obsNo(X).
6  visited(X) :- anyObsYes(X).
7
8  connected(X, X) :- visited(X).
9  connected(X, Z) :- visited(X), edge(X, Y), connected(Y, Z).
10
11 :- order(X, Y), not connected(X, Y).
12 :- entry(E), visited(X), not connected(E, X).
13 :- crash(C), visited(X), not connected(X, C).
14 :- anyObsYes(O), visited(X),
15    not connected(X, O), not connected(O, X).
```

Figure 3. Answer set programming base formulation

encode basic constraints over nodes visited on legal paths: we must visit the program's entry (line 3) and the crash point (line 4), we must not visit *obsNo* entries (line 5), and we must visit all *anyObsYes* entries (line 6). Next, the logic program defines connectivity: a visited node is always connected to itself (line 8), and two nodes are inductively connected if we can cross one edge from the visited start node to an intermediate node, and then connect this intermediate node to the destination (line 9). Note that two nodes are only connected if they are connected on paths corresponding to the generated model. In other words, connected does not express general properties of G, but, rather, properties of a set of paths through G (with identical visited sets).

The final rules in lines 11–15 express other relationships between path connectivity and our observations. Each of these "headless" rules specifies a set of forbidden configurations; that is, none of the stated conditions are satisfied in any valid model. Line 11 states that ordered *obsYes* entries must be connected. The next two rules assert that all visited nodes are connected to both the program entry (line 12) and the crash point (line 13). The final rule on lines 14–15 enforces paths between each *obsYes* sequence: every visited node must be connected with every *anyObsYes* entry.

Given these constraints, we can compute *exeYes*, *exeNo*, and *maybe* by determining if each $n \in G$.nodes may or may not be visited in generated models. Recall that, in the worst case, our program generates an exponential number of models. However, rather than iterating over models, we instead iteratively check satisfiability for both visiting and not visiting each $n \in G$.nodes. Specifically, we begin with the complete, conjoined set of constraints C from a failure report and Figure 3. Then, for each $n \in G$.nodes, we form the augmented set of constraints: (i) C_n^{yes} by adding the constraint "visited(n)." to C, and (ii) C_n^{no} by adding the constraint "–visited(n)." to C. If C_n^{yes} is unsatisfiable, then $n \in$ *exeNo*. If C_n^{no} is unsatisfiable, $n \in$ *exeYes*. If both C_n^{yes} and C_n^{no} are satisfiable, then $n \in$ *maybe*.

Table 1. Resource usage. Entries labeled ">5 m" ran out of memory before hitting our 3 hour timeout.

		Time				Memory			
		crash only		csi-cc info		crash only		csi-cc info	
Subject	LoC	FSA	ASP	FSA	ASP	FSA	ASP	FSA	ASP
Figure 1	13	<1 s	<1 s	<1 s	<1 s	19 MB	18 MB	19 MB	18 MB
schedule	413	4.1 s	53 m	3.5 s	34 m	41 MB	218 MB	41 MB	197 MB
ccrypt	5280	3.0 m	>3 h	>5 m	>3 h	79 MB	≈12 GB	≥32 GB	≈9.5 GB

As in Section 2.1, our approach is intraprocedural, but, if G is the interprocedural CFG of P, we can obtain a context-insensitive interprocedural result by applying our analysis to the CFG (with the entry predicate attached to the entry node for the main function). Prior work [18, 20] uses other logic-programming paradigms (particularly Datalog) for context-sensitive program analyses. That work provides the basis for extending our analysis to be context-sensitive.

3. Evaluation

We performed a preliminary evaluation to assess the feasibility of the techniques described in Section 2 with two primary goals. First, we wanted to check our claim that our two formulations are equivalent. Second, we wanted to investigate the memory and running-time costs of computing *exeYes*, *exeNo*, and *maybe* using each technique.

For our automaton-based formulation from Section 2.1, we used the OpenFst library [2]. We chose the Gringo grounder [10] with its accompanying Clasp [11] SAT solver for the implementation of our ASP formulation from Section 2.2. CodeSurfer 2.2p0 [4] produces our CFGs.

We used three applications: a toy program conforming to the skeleton code from Figure 1, schedule (a priority scheduler from the Siemens benchmark suite), and ccrypt (an encryption program). We hand-crafted the toy program; schedule is taken from the Software-artifact Infrastructure Repository [19]; ccrypt is a real, released program. The defect in schedule was seeded, while the defect in ccrypt arose naturally. For each application, we ran our analysis for one failing run from the program's test suite. First, we used failure reports consisting of only the crash location. Second, we used failure reports containing call-site coverage information as gathered by the csi-cc tool developed by Ohmann and Liblit [16]. This tool gathers unordered coverage data with no missed observations; the tool instruments all call sites, and each call site appears either in *obsNo* or as a single-element sequence in *obsYes*. All experiments were run on a 3.1 GHz quad-core Intel Core i5 with 32 GB of RAM running Red Hat Enterprise Linux 6.6. For these experiments, we used interprocedural CFGs, but context-insensitive analyses.

First, we verified that our FSA and ASP implementations produced the same *exeYes*, *exeNo*, and *maybe* results for the failure reports from our test cases. In all subjects that completed in under three hours, we obtained identical results. We also manually verified that these results matched the expected output. Next, we evaluated the time and memory usage for computing *exeYes*, *exeNo*, and *maybe* based on the failure reports for each test case; the results are shown in Table 1. We measured real time and maximum resident memory size. We report the mean of three runs, though all trials were fairly consistent. For small cases such as Figure 1, both approaches work well. For larger programs, the automaton implementation was clearly faster in these experiments; however, with csi-cc profiling information, we observe a substantial increase in FSA size when adding the *obsYes* constraints for the ccrypt benchmark. Conversely, our ASP implementation is currently slower and hits our time threshold for ccrypt; however, all our test cases fit in system memory. Thus, had we allowed substantially more time, our ASP implementation would likely have completed all tests without running out of memory.

Our current implementations are very naïve, and we made no efforts to optimize our techniques; thus, we suspect that these results could be substantially improved. Nevertheless, these experiments serve to highlight the strengths and weaknesses of each approach, and help to guide future research.

4. Related Work

Prior work on program coverage adds instrumentation selectively, leveraging the fact that observed execution at one program point may imply execution elsewhere [1, 5, 22]. Rather than computing these implications during instrumentation, our approach solves exactly the problem of computing them after-the-fact. Also, prior approaches produce instrumentation that ensures full coverage information during execution. Our technique allows for incomplete data: observations may leave some information unknown.

Prior work has used symbolic execution to replay failing executions [8, 13, 27]. While our work similarly attempts to recover execution data from failing runs, we solve a different problem. We recover information matching *all* runs consistent with traced data, while replay techniques synthesize *one* complete run consistent with traced data.

Nevertheless, we could potentially adapt existing symbolic-execution-based approaches to eliminate some *maybe* paths that are infeasible based on data values observed at the time

of the crash. Yuan et al. [25, 26] apply this idea to run-time logs. They use the locations and content of log messages to infer *must*, *may*, and *must-not* paths through a program. Our work considers log messages as one example of possible probe points, but also handles unordered trace data, such as that produced by Ohmann and Liblit [16]. We also solve a different problem: we mark each program *statement* as *must*, *may*, or *must-not*, while Yuan et al. infer partial execution *paths* that must, may, or must not have executed.

The incomplete-observations problem closely resembles the problem of evaluating a formula with respect to a partial (i.e., incomplete) model. For the latter, the best possible result is given by the formula's *supervaluational meaning* [7, 17, 23, 24]. In our context, the encoding of the failure report plays the role of the incomplete model, and a probe—e.g., the automaton for $obsYes_n$—plays the role of the formula to be evaluated. We are populating *exeYes*, *exeNo*, and *maybe* according to the supervaluational meaning of these queries.

5. Conclusions and Future Work

In this paper, we present an approach to derive unobserved execution information from failing applications with incomplete observation data. We present two distinct encodings of failure reports, and show how these encodings can be applied as solutions to the incomplete-observations problem.

We are currently evaluating these two techniques more extensively. Neither approach currently supports interprocedural context-sensitivity; we plan to investigate methods by which this could be added. The present failure model does not cover the common case of a statement log without missing observations. We could model such situations by modifying *obsYes* such that, for all paths r through G, each occurrence of an $o \in anyObsYes$ in r must correspond to an entry in each *obsYes* sequence. This requires only minor implementation changes. For example, in Figure 2d, we would merely relabel each self-loop as "$\Sigma - \{B, C\}$".

Acknowledgments

This research was supported in part by NSF grants CCF-0904371, CCF-0953478, CCF-1217582, and CCF-1420866. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] H. Agrawal. Dominators, super blocks, and program coverage. In *POPL*, 1994.

[2] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *CIAA*, 2007.

[3] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[4] P. Anderson, T. W. Reps, and T. Teitelbaum. Design and implementation of a fine-grained software inspection tool. *IEEE Trans. Software Eng.*, 29(8):721–733, 2003.

[5] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *TOPLAS*, 1994.

[6] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, 1996.

[7] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *CONCUR*, 2000.

[8] Y. Cao, H. Zhang, and S. Ding. SymCrash: selective recording for reproducing crashes. In *ASE*, 2014.

[9] B. Gauf and E. Dustin. The case for automated software testing. *Journal of Software Technology*, 10(3):29–34, 2007.

[10] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In *LPNMR*, 2011.

[11] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187: 52–89, 2012.

[12] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1):4–12, Jan. 2002.

[13] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *ICSE*, 2012.

[14] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[15] V. W. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. *CoRR*, 1998.

[16] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *ASE*, 2013.

[17] T. Reps, A. Loginov, and M. Sagiv. Semantic minimization of 3-valued propositional formulae. In *LICS*, 2002.

[18] T. W. Reps. Solving demand versions of interprocedural analysis problems. In *CC*, 1994.

[19] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do. Software–artifact infrastructure repository, Sept. 2006.

[20] Y. Smaragdakis and M. Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog 2010. Revised Selected Papers*, 2011.

[21] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *NIST, RTI Project*, 7007(011), 2002.

[22] M. M. Tikir and J. K. Hollingsworth. Efficient online computation of statement coverage. *Journal of Systems and Software*, 78(2), 2005.

[23] B. van Fraassen. Singular terms, truth-value gaps, and free logic. *J. Phil.*, 63(17):481–495, Sept. 1966.

[24] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *Trans. on Comp. Logic*, 8(1), 2007.

[25] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*, 2010.

[26] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *ASPLOS*, 2011.

[27] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, 2010.