Program Reduction: A Comprehensive Survey

YONGQIANG TIAN, Monash University, Australia
ZHENYANG XU, University of Waterloo, Canada
HONGXU XU, University of Waterloo, Canada
XINTONG ZHOU, University of Waterloo, Canada
CHUNHAO LIAO, University of Waterloo, Canada
YIWEN DONG, University of Waterloo, Canada
MENGXIAO ZHANG, University of Waterloo, Canada
CHENGNIAN SUN, University of Waterloo, Canada

Program reduction transforms a failure-inducing program into a smaller program that preserves the failure of interest. The design space of modern reducers extends well beyond deletion plus an oracle that decides whether a candidate still triggers the failure: they differ in program representation, transformation space, search and guidance, stopping criteria, oracle engineering, and validity control. This survey covers the families that populate that space: list-based Delta Debugging, hierarchical, syntax-guided, language-specific semantic, search-prioritized, and large language model-aided reducers. These families coexist rather than supersede one another, because each resolves those design choices differently. We compare how these choices affect portability, effectiveness, efficiency, and practical utility, and examine open challenges in validity, oracle cost, benchmark design, and reproducible evaluation.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Program Reduction, Delta Debugging, Survey

ACM Reference Format:
Yongqiang Tian, Zhenyang Xu, Hongxu Xu, Xintong Zhou, Chunhao Liao, Yiwen Dong, Mengxiao Zhang, and Chengnian Sun. 2026. Program Reduction: A Comprehensive Survey. 1, 1 (September 2026), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When language processors, such as compilers, type checkers, and static analyzers, exhibit unexpected behaviors like crashes, miscompilations [76, 106], or incorrect analysis results [57], developers rely on concrete inputs to reproduce and debug the underlying defect. An input, such as a source program, that reliably triggers the bug is called a *failure-inducing program*. In practice, the failure often depends on only a tiny fragment of an otherwise large input [85, 98, 100]. For instance, a compiler crash may be triggered by a single syntactic pattern [77], a miscompilation by the interaction of a few statements [98], and an incorrect warning by a narrow code idiom [75]. Yet, when diagnosing the issue, developers are initially confronted with the full program and all its incidental complexity. Manually separating the failure-inducing fragment from irrelevant code is laborious and does not scale to the massive inputs produced by modern test generators and fuzzers [47, 51, 58, 106]. Furthermore, a failure whose root cause remains buried in a large program is harder to report, reproduce, and diagnose [8, 39, 109, 111].

Authors' addresses: Yongqiang Tian, yongqiang.tian@monash.edu, Monash University, Melbourne, Victoria, Australia; Zhenyang Xu, zhenyang.xu@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Hongxu Xu, hongxu.xu@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Xintong Zhou, x27zhou@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Chunhao Liao, chunhao.liao@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Yiwen Dong, yiwen.dong@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Mengxiao Zhang, max.zhang@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada.

To address this challenge, automated techniques known as *program reduction* transform a failure-inducing program into a smaller one that still triggers the same failure. The tools that perform this transformation are called *reducers*. In this survey, we use the term *program* broadly for any structured failure-inducing artifact, ranging from the source code of a general-purpose language (e.g., a compiler bug-triggering program [66]) to a declarative formula like a Satisfiability Modulo Theories (SMT) instance [45]. Reducers typically operate iteratively; at each step, they produce a *candidate* (i.e., a proposed smaller variant of the program). The automated check that determines whether a candidate still triggers the target failure is called the *oracle*.[1]

Program reduction has become an important practical technique in modern software testing and debugging. By automatically discarding irrelevant code [66, 113], reduction directly mitigates the difficulty of reporting and diagnosing bugs hidden in large programs. For example, test programs generated by compiler fuzzers such as Csmith [106] usually range from 37 to 297 KB [66]. C-Reduce, a program reducer for the C language, reduces these bug-triggering inputs to a median size below 0.5 KB, more than 25× smaller than line-based Delta Debugging, producing outputs suitable for direct inclusion in bug reports [66]. A large-scale study of more than 50,000 GCC and LLVM bugs likewise shows that the bug-revealing test cases retained in regression suites are usually small: 80% have fewer than 45 lines of code [77]. This is reinforced by project policy: both GCC and LLVM explicitly ask bug reporters to submit reduced failure-inducing programs [18, 52]. While these specific numbers from C/C++ compilers are merely illustrative, they make concrete why high-quality reduction matters in practice. Consequently, tools such as Lithium [67], C-Reduce [66], and Perses [78] are widely used in compiler testing and fuzzing pipelines, and the same basic methodology has been extended to proof-assistant scripts [25], solver inputs [45], and other structured failure-inducing programs.

The field of program reduction has diversified methodologically since its founding formulations. Early work introduced Delta Debugging (DD) for isolating failure-inducing differences and *ddmin* for single-input minimization over list-based representations such as characters, tokens, and lines [110, 113]. The *ddmin* algorithm partitions the failure-inducing input and searches for a minimal failure-inducing subset. Hierarchical Delta Debugging (HDD) then introduced tree-structured reasoning over nested inputs [56]. Later systems pushed further by exploiting program structure. GTR generalizes tree-based reduction beyond plain deletion with *tree templates* that delete a node or substitute it with one of its children [30]. Perses was the first reducer to use the language's formal grammar to guide reduction, keeping every candidate syntactically valid and enabling moves that contiguous deletion cannot express, such as hoisting (i.e., replacing a subtree with a same-type descendant) [78]. Subsequent work expanded the design space through richer transformations [102, 105], stronger validity control [45], more deliberate search guidance [94], more explicit attention to oracle cost [86], and growing interest in canonicalization for deduplication, i.e., collapsing reduced reports that share one underlying bug [103]. Most recently, large language model-aided approaches have begun exploring model-proposed rewrites beyond what deletion [113] and transformations over parse trees, such as hoisting and tree templates [30, 78], can easily generate, while still relying on reducer and oracle checks to reject invalid or unhelpful candidates [114]. The resulting history is not a linear departure from Delta Debugging, but a progressive response to recurring limitations of deletion-only reduction, including malformed intermediates [56, 78], shallow local minima [89, 104], weak validity guarantees [66], and wasted oracle queries [86, 94].

This methodological expansion raises the question of how program-reduction techniques should be designed, compared, and evaluated. The modern literature is more than a simple sequence of

[1]The same concept also appears in the literature as the *interestingness test* [113], *property test* [29], or *interestingness predicate*. We use *oracle* as the primary term throughout this survey.

refinements, because reducers now differ along interacting design choices: program representation, transformation space, search and guidance, stopping criteria, oracle engineering, and validity control. The field therefore needs a comparative structure that explains these design choices and trade-offs, brings prior work into a common view, and helps surface future research directions.

We present *a systematic survey of program reduction.* We assembled a candidate set of peer-reviewed program-reduction research through keyword search, backward citation chasing, and forward citation chasing. Three authors then reviewed the candidate papers and classified their topics, scope, and methodological role to determine which papers belonged to the core field of failure-inducing program reduction. This process yielded a main corpus of 67 papers for detailed analysis, together with adjacent and excluded sets that clarify the field's boundaries. We separated work on collection-level reduction, such as test-suite minimization, from work on reducing a single failure-inducing program. We also treated debloating [1], trace or execution reduction [7], configuration and systems debugging, AI-model simplification [63], and uses of DD [113] or HDD [56] without a substantive methodological contribution to program reduction as adjacent context rather than as part of the main corpus. To make the design space of modern reducers comparable, we propose *an abstract reduction framework.* The framework compares reducers through the choices that structure the reduction process: how the current program is represented, how candidate variants are generated and constrained for validity, how search is prioritized, how the oracle is used, how state is updated after success or failure, and how stopping is defined. Making those choices explicit helps explain why similar reducers can behave quite differently in practice, and why cross-paper comparisons are easily distorted by differences in these design choices or in evaluation setup.

With this framework and comparative analysis, we obtain five survey-level findings, each developed in the following sections. *First,* the field is not a linear progression toward a single dominant approach: list-based, hierarchical, syntax-guided, language-specific semantic, search-prioritized, and large language model-aided reducers coexist because they reflect different trade-offs in portability, structural precision, oracle cost, and practical utility, as § 5 develops. *Second,* much of the field's technical diversification stems from the recurring limitations of deletion-only reduction noted above rather than from independent innovation, as § 6 traces family by family. *Third,* cross-paper comparison is difficult not because metrics are absent, but because the meaningful object of comparison extends beyond the reducer itself to the program class, the oracle wrapper (the script that invokes the oracle), the build or replay environment, and the stopping criterion, as § 8.3.2 documents. *Fourth,* stronger guarantees require narrower assumptions: gains in validity preservation, semantic faithfulness (preserving the program's runtime semantics, not just its syntax), or canonicalization typically entail higher engineering cost, deeper domain expertise, lower portability, or more expensive validation, as §§ 5 and 6 show. *Fifth,* the property a reducer preserves, not its deletion algorithm, sets the objective, the standard of validity, and what counts as a correct result; failure reproduction is the most common instance of property-preserving minimization rather than its definition, as §§ 7 and 8 develop.

The survey makes the following contributions:

- an abstract reduction framework that explains how modern reducers are constructed from recurring design choices (program representation, transformation space, search and guidance, stopping criteria, oracle engineering, and validity control), providing a systems-level vocabulary for understanding and building reducers;
- a comparative analysis that organizes reducers along recurring comparative dimensions (representation, language knowledge, guidance strategy, and relational scope), explaining how those same design choices create meaningful differences across the literature;
- a transformation-space taxonomy that groups the surveyed transformations into four families ordered by the language knowledge each requires (deletion and pruning, structural rewriting,

lexical canonicalization, and semantic transformation), with an explicit criterion separating deletion-based transformations from rewriting;
- an application-domain analysis that pairs each domain with the object it reduces, the property it preserves, and the objective that follows;
- a synthesis of open challenges organized as what a reducer should preserve (canonical form, validity, and the preserved property itself), how the reduction is reached (oracle cost, learned and large language model-aided guidance, and the domain-expertise barrier), and whether a result transfers (robustness across contexts and comparable, reproducible evaluation).

The rest of the paper develops these points in turn. § 2 explains corpus construction and scope, and § 3 introduces the prerequisite concepts. §§ 4–6 develop the framework, comparative dimensions, and transformation space. § 7 surveys application domains. § 8 then sets out open problems, including how reducers should be compared and evaluated, and § 9 concludes.

2 SURVEY METHODOLOGY

This survey is based on a candidate set of papers assembled through keyword search, backward citation chasing, and forward citation chasing. We issued queries centered on *program reduction*, *delta debugging*, *testcase reduction*, and *input minimization* against IEEE Xplore, ACM Digital Library, and Scopus in April 2026. We then expanded the candidate set through backward and forward citation chasing, yielding 162 candidate papers. The three authors then reviewed the collected publications and classified their topics, scope, and methodological role to determine which papers belonged to the core field of failure-inducing program reduction. Disagreements over inclusion decisions were resolved through author discussion until consensus was reached. Ambiguous cases were assigned to the adjacent corpus when the paper's primary contribution lay outside failure-inducing program reduction itself.

In this survey, we distinguish the *candidate set* from the *main corpus*. The candidate set is intentionally broader, covering the field's boundaries and related literatures that employ similar reduction ideas. The main corpus is defined by the scope of *failure-inducing program reduction*: the automated reduction of a single failure-inducing program (such as source code, a solver input, or a proof-assistant script) that triggers a reproducible failure under an automated oracle, while preserving that failure. The term *failure-inducing program* is used throughout to refer to this class of structured inputs [66, 78]. Applying this criterion, the candidate set partitions into 67 main-corpus papers, 82 adjacent papers, and 13 out-of-scope papers, as shown in Figure 1.

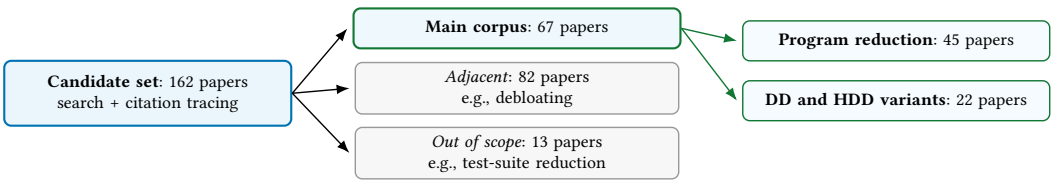


Fig. 1. Organization of the surveyed literature. The candidate set partitions into the main corpus, adjacent areas, and out-of-scope papers based on each paper's contribution to failure-inducing program reduction; the main corpus further divides into dedicated program-reduction papers and algorithmic variants of DD/HDD.

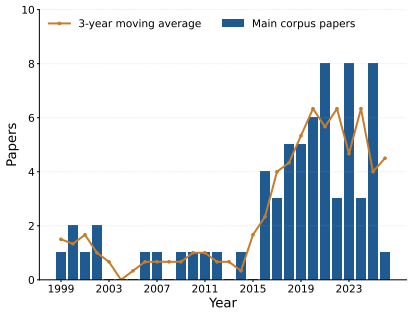
A paper enters the main corpus when it makes a direct contribution to failure-inducing program reduction, in either of two ways. *First,* it contributes a method for reducing a single failure-inducing program, whatever the paper calls it ("program reduction," "test-case reduction," or "input minimization"), as long as the reduced artifact is itself a failure-inducing program whose syntactic or constraint structure the reduction must respect, such as a grammar, well-formedness rules,

or theory constraints. This covers reducers such as Perses [78] together with compiler-bug [66], typed-language [68], solver-input [45], and proof-assistant [25] reduction published under the "test case" label. *Second,* it extends the DD or HDD algorithmic framework and its main contribution changes the reduction algorithm or its guarantees, rather than instantiating DD or HDD unchanged in a new pipeline, as with Probabilistic Delta Debugging (ProbDD) and Weighted Delta Debugging (WDD) [94, 120].

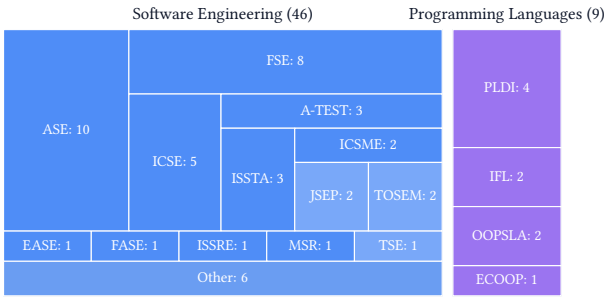
We do *not* place a paper in the main corpus merely because it mentions or uses DD. Papers that apply DD inside broader debugging [108, 109] or systems debugging [119] are treated as adjacent unless they make a direct methodological contribution to program reduction itself. Beyond that rule, the adjacent and out-of-scope bands in Figure 1 are separated by what a work reduces and whether it contributes a reduction method.

We treat a line of work as an *adjacent area* when reduction is still central but the reduced object is not a single failure-inducing program. This happens on two grounds. The artifact may not be program-like, i.e., not a static artifact that a language processor parses and evaluates but a record or setting of dynamic behavior, as in trace or execution reduction [7, 27] and in configuration or systems debugging, where the reduced object is a deployment configuration or a system-level failure scenario [53]. Alternatively, the goal may be deployment or interpretability rather than reproducing a failure, as in software debloating [1] and AI-model simplification [63].

We treat a line of work as *out of scope* on either of two grounds. The first is that its object is not a single input. For example, test-suite minimization and test-set reduction select a representative subset of a set of tests that preserves a suite-level property such as code coverage or fault-detection capability, rather than performing the oracle-driven candidate testing that defines program reduction [72]. The second is that papers make no reduction contribution, as with tutorials.



(a) Publication timeline with a moving average to smooth short-term variation.



(b) Publication venues. The venue names are listed individually when ranked CORE A* or A or contributing at least two papers

Fig. 2. Publication timeline and venues of the main corpus.

With the corpus fixed, Figure 2a complements Figure 1 by tracing the publication history of the main corpus from 1999 to 2026. The timeline shows steady foundational activity before 2016, followed by a denser growth period that accounts for the majority of the 67 main-corpus papers. This expansion motivates the survey's emphasis on comparison along the dimensions of § 5 rather than chronology alone: the acceleration of publication activity reflects increasing methodological diversity rather than linear extension of earlier work. Figure 2b shows the software-engineering and programming-languages publication venues of the main corpus; the largest groups are concentrated in software engineering, with a smaller but consistent presence in programming languages.

3 FOUNDATIONAL CONCEPTS IN PROGRAM REDUCTION

This section introduces the conceptual foundations that recur throughout the program-reduction literature. It formulates program reduction as a failure-preserving search problem, explains how Delta Debugging (DD) and its minimizing algorithm *ddmin* instantiate the canonical baseline for that problem, and introduces the criteria by which reducers are evaluated throughout the survey.

3.1 The Program Reduction Problem

In this survey, program reduction is the task of transforming a failure-inducing program into a smaller variant that still preserves the behavior of interest, i.e., it triggers the same failure as the original program. As in the introduction, the term *program* is used broadly for the structured failure-inducing artifact being reduced, ranging from source code of a general-purpose language to a declarative formula such as an SMT instance [45]. In debugging settings, the objective is to isolate the structural core that continues to reproduce the original fault, thereby making the failure easier to inspect, explain, and communicate [102, 112, 113]. This formulation is broader than line-level deletion, but it excludes the general reduction of arbitrary system states or execution traces.

A reducer addresses the following question: *how small can a failure-inducing program become while still triggering the same failure?* To answer it, the reducer repeatedly proposes a simpler candidate and checks whether the target failure still occurs. That automated check is the *oracle*. For example, it may rerun a compiler crash [66], recheck a type error [69], replay a recorded trace [4], or check any other failure condition of interest.

Formally, the oracle is modeled as a predicate ψ. Given a candidate program P', the oracle returns $\psi(P') = \text{true}$ if P' preserves the property of interest, that is, still exhibits the target failure, and $\psi(P') = \text{false}$ otherwise. For brevity, we write "$\psi(P')$ holds" for $\psi(P') = \text{true}$. Let P denote the original failure-inducing program, for which $\psi(P) = \text{true}$ by assumption. Program reduction searches for a variant P' such that $\psi(P')$ holds and $|P'| < |P|$, where $|\cdot|$ denotes the reducer's chosen size measure [24, 113]. The most common guarantee in this setting is **1-minimality**: no single unit (such as a line, token, or tree node) can be removed from P' while keeping ψ true [24, 113]. Because the space of failure-preserving variants grows combinatorially with input size, practical reducers cannot rely on exhaustive search and must instead use heuristics, partitioning strategies, and domain constraints to keep the problem tractable [24, 49, 78, 113]. Consequently, 1-minimality is an inherently local guarantee constrained by the reducer's chosen representation and transformations; a richer transformation set could often simplify the result further [92]. For tree-structured representations, the same guarantee instantiates as **1-tree-minimality**, where no single tree node can be removed from the tree representation of P' while preserving the failure [56].

3.2 The Delta Debugging Foundation

Delta Debugging (DD) is the general framework for isolating failure-inducing differences. Given an input that exhibits the target failure ($\psi = \text{true}$) and a reference input that does not ($\psi = \text{false}$), it minimizes the difference (the *delta*) between them [110]. The *ddmin* algorithm is the special case of DD in which the reference input is empty, so minimizing the difference reduces to minimizing the failure-inducing input itself (i.e., simplification). Specifically, *ddmin* operates on that input decomposed into a flat list of configuration elements (a *list-based* representation) and searches for a smaller configuration on which ψ still holds [113]. DD generalizes *ddmin* by admitting a non-empty reference input, which it additionally maximizes to isolate a minimal failure-inducing difference [113]. *ddmin* is foundational as the first widely reusable formulation of failure-preserving search, turning the general problem into a concrete oracle-guided procedure.

Algorithm 1: The *ddmin* algorithm over a flat configuration

Input : a configuration C for which $\psi(C)$ holds
Output: a 1-minimal sub-configuration that still satisfies ψ

Function ddmin(C):
  $k \leftarrow 2$ // granularity: number of subsets
  while $|C| > 1$ **do**
    partition C into subsets $\Delta_1, \ldots, \Delta_k$ of nearly equal size
    if $\exists\, i :\ \psi(\Delta_i)$ *holds* **then**
      $C \leftarrow \Delta_i; \quad k \leftarrow 2$ // Step 1 (subset test): reduce to the subset
    else if $\exists\, i :\ \psi(C \setminus \Delta_i)$ *holds* **then**
      $C \leftarrow C \setminus \Delta_i; \quad k \leftarrow \max(k-1,\, 2)$ // Step 2 (complement test): reduce to the complement
    else if $k < |C|$ **then**
      $k \leftarrow \min(2k,\, |C|)$ // Step 3 (refine granularity): increase k
    else
      break // Step 3: 1-minimal under the current decomposition
  return C

In the classical formulation of *ddmin* (Algorithm 1), the initial program is *decomposed* into a configuration $C = \{c_1, \ldots, c_n\}$ of elements with $\psi(C) = \text{true}$ for this failure-inducing starting point. Given C and ψ, the algorithm *partitions* C into k subsets, where k is the *granularity* (initially $k = 2$), and proceeds in three steps.
Step 1 (subset test): for each subset S of the partition, test whether $\psi(S)$ holds. If so, recurse into S as the new configuration with $k = 2$; otherwise, go to Step 2.
Step 2 (complement test): for each subset S, test whether $\psi(C \setminus S)$ holds. If so, continue from the complement with $k = k - 1$; otherwise, go to Step 3.
Step 3 (refine granularity): if every subset already contains a single element, terminate; otherwise, double k (up to the number of remaining elements) and return to Step 1.

The abstract search for a smaller P' thus becomes a concrete sequence of oracle queries over candidate configurations, and successful termination yields a result that is 1-minimal relative to the chosen decomposition [24, 113]. The vocabulary of configurations, interestingness tests, and cause-effect isolation (isolating the minimal failure-inducing difference between a failing input and a reference input) that still shapes modern reduction work was introduced in the original DD framework [110] and carried forward into *ddmin*'s single-input formulation [113].

The *ddmin* algorithm provides two theoretical guarantees: its result is *1-minimal* (§ 3.1), and the number of oracle tests it performs is bounded, as developed in the complexity discussion below. These guarantees rest on three assumptions about the search space, namely *consistency*, *unambiguity*, and *monotonicity* [110, 113].

Consistency assumes the oracle returns a determinate verdict for every candidate. In Delta Debugging's classical formulation, which follows the POSIX 1003.3 standard for test outcomes [36], the test function can also return a third outcome, *unresolved*, and consistency assumes that outcome never occurs [110, 113]. The unresolved outcome marks candidates on which the preserved property is not even defined. For instance, deleting a variable's declaration from a C program that crashes the compiler yields a candidate that no longer compiles, so the question of whether it exhibits the target crash has no meaningful answer [56, 78]. Practical oracles instead return true or false for every candidate, as the definition of ψ above does. Any candidate that does not exhibit the target failure, including one on which the property is not even defined, is simply mapped to false. The third value thus disappears by convention, but the distinction it expressed still has to be made somewhere. With only true and false available, telling the target failure apart from every other outcome, build errors included, becomes entirely the oracle's job. Consider again the candidate that no longer compiles. An oracle that merely checks for a nonzero exit status treats its build error

like the target crash and returns true. The *ddmin* algorithm then keeps the deletion that broke the program, and the reduction can drift, step by step, toward a program that triggers a different failure than the one under investigation. Moreover, mapping such candidates to false settles their verdict but not their cost, since the reducer still generates and tests them. Each one consumes an oracle query just to be rejected, a waste that validity control (§ 4) exists to avoid. Beyond these two costs, consistency can also fail outright when a flaky compiler or environment returns different verdicts for the same candidate across runs [70].

Unambiguity assumes a failure has a single cause rather than two independent ones, so *ddmin* need not look for a second cause once it has found one [110]. It breaks when an input has two independent fault sites that each reproduce the failure on their own, say two unrelated functions in one C file that each crash the compiler. Removing either one still leaves a failure-inducing program, so which fault *ddmin* keeps depends on its traversal order. This is harmless for reduction, since either cause gives a valid smaller program, but it matters for diagnosis and canonicality, i.e., equivalent failures reducing to a stable, comparable form, where a single representative is wanted.

Monotonicity assumes that once ψ holds for a configuration, it also holds for every superset, and equivalently that no subset of a configuration with ψ = false yields true [110]. It breaks when deletions interfere. Suppose two elements are jointly removable, so the failure survives once both are gone, yet removing either alone makes it disappear. For the configuration with both elements removed, ψ is thus true, while for its superset with only one of them removed it is false, violating monotonicity. However, *ddmin* guarantees only 1-minimality, meaning no single element can be dropped while preserving the failure. That already holds here, so *ddmin* can stop with both elements present, even though removing the two together would give a smaller program that still exhibits the failure. Such interference leaves deletion-based search at a local minimum well above the true minimum, i.e., the smallest failure-inducing variant reachable by any sequence of removals, which is what the search-guidance and richer-transformation work in § 6 sets out to escape.

The three assumptions protect distinct aspects of *ddmin*'s guarantees. Consistency keeps the reduced program faithful to the target failure, monotonicity keeps its 1-minimal result close to the true minimum, and unambiguity keeps the search economical and its result canonical.

The time complexity of *ddmin* exposes a recurring limitation of deletion-based search. In favorable cases, when failure-inducing structure is cleanly isolated and the oracle consistently evaluates coarse partitions, the search terminates in $O(\log n)$ oracle calls [113]. In the worst case, however, oracle cost grows quadratically in the number of candidate elements because interacting failure causes force repeated repartitioning and repeated tests of overlapping subsets and complements [110, 113]. This gap between favorable and adversarial cases is important for the later literature: many subsequent techniques can be read as attempts either to reduce the number of costly oracle calls [16, 20, 86, 94] or to improve the quality of the local minimum reached by deletion-based search [56, 78, 89].

Beyond cost, the significance of *ddmin* lies in the architectural abstraction it established. It demonstrated that reduction can be driven by an external oracle without requiring a semantic model of the target language, making the framework applicable across many program classes and domains [24]. This oracle-centered abstraction remains visible in modern reducers: syntax-guided [78] and search-prioritized [94] reducers, and systems with stronger validity control [102], still test candidate simplifications against an oracle. At the same time, arbitrary subsetting readily produces invalid intermediates when programs are governed by strong syntactic or semantic dependencies, making the search less efficient and less informative [56, 102]. Many modern systems therefore retain *ddmin*'s partition-based search while surrounding it with richer representations [56], stronger validity control [78], and broader transformation operators [102, 104].

3.3 Evaluation Criteria

Reducers can be optimized and compared from several perspectives. From the literature, we identify three main criteria: effectiveness, efficiency, and practical utility. These are not independent, and improving one often comes at the expense of another.

Effectiveness captures how small the result becomes. The most common formal guarantee is 1-minimality (§ 3.1) [24, 113], or 1-tree-minimality for tree-structured representations [56], where no single unit can be removed while preserving the failure. Once a reducer reaches that point, the resulting size is reported in absolute terms and as a reduction ratio, i.e., the final size divided by the size of the original input or of a baseline reducer's output [46, 66]. Effectiveness is the most visible criterion and the one classical reducers were built to optimize, yet it is also the easiest to over-pursue at the expense of the other two.

Efficiency captures the cost of reaching that result. The dominant cost is the oracle, since every candidate must be checked against it. When that check is a compiler run, a solver call, or a test workflow, the running time scales with the number of oracle queries rather than the number of edits. Efficiency is thus reported as the oracle-query count and as wall-clock time under a stated oracle, and much recent work targets it through prioritized search [94], caching [86], and parallelism [93].

Practical utility captures how useful the reduced program remains afterward, and unlike the first two it has no single number. It spans validity (whether the result is still parsable, compilable, replayable, or well-sorted for its target, the SMT analogue of type-correct) [45, 66], canonicality (whether equivalent failures reduce to a stable, comparable form that supports deduplication and triage) [13, 103], and diagnostic value (whether the result still carries the context a developer needs to understand the failure) [26, 116]. A result that scores well on effectiveness can score poorly here, which is why aggressive minimization is not always an improvement [66, 116].

4 THE PROGRAM REDUCTION WORKFLOW

Modern program reducers have diverged substantially from the classical *ddmin* baseline, varying simultaneously in syntax guidance [78], transformation frameworks [105], lexical canonicalization [103], search prioritization [94], caching [86], and execution-aware reduction [4]. Reviewing reducers in historical development order is therefore an insufficient basis for comparison. This section introduces the paper's main comparative contribution: an abstract reduction framework organized around six recurring design choices. These design choices are program representation, transformation space, search and guidance, state update and stopping, oracle engineering, and validity control; together they constitute the *workflow* of a reducer.

4.1 An Abstract Reduction Framework

The abstract reduction framework we propose for comparing program reducers is presented in Algorithm 2. The framework captures a recurring pattern: a reducer chooses a working representation of the current program, generates candidates permitted under that representation, tests those candidates with the oracle, updates its state based on success or failure, and stops when the current program is irreducible under its representation and transformation space. A *transformation* is an operator schema, such as deleting a subset or hoisting a node, whereas a *candidate* is the concrete program obtained by binding one operator to a specific location, so a single transformation typically yields many candidates. The framework therefore separates `applicable_transformations`, which fixes the operators permitted under the current representation, from `generate_candidates`, which instantiates those operators into concrete candidate programs under the search policy. The split reflects how the surveyed reducers differ. ProbDD and WDD keep the transformation space of *ddmin*, deletion only, and change only how candidates are selected and ordered [94, 120], whereas

Algorithm 2: An Abstract Reduction Framework.

Input : a failure-inducing program P and an interestingness predicate ψ
Output: a reduced program P' such that $\psi(P')$ still holds

Function reduce *(P, ψ)*:
  $P' \leftarrow$ represent(P) // Program Representation
  $\sigma \leftarrow$ initialize_state(P', ψ) // State Update and Stopping
  while not should_stop *(P′, σ)* **do**
    $\mathcal{T} \leftarrow$ applicable_transformations(P', σ) // Transformation Space
    $\mathcal{K} \leftarrow$ generate_candidates$(P', \mathcal{T}, \sigma)$
    $\mathcal{K} \leftarrow$ prioritize$(\mathcal{K}, P', \sigma)$ // Search and Guidance
    reduced $\leftarrow$ false
    foreach $p \in \mathcal{K}$ **do**
      if test *(p, ψ)* **then** // Oracle Engineering
        $(P', \sigma) \leftarrow$ on_success(P', p, σ)
        reduced $\leftarrow$ true
        break
    if not *reduced* **then** $(P', \sigma) \leftarrow$ on_failure(P', σ)
  return P'

Perses and GTR widen the operator set while keeping a systematic traversal [30, 78]. A framework that merged the two steps could not state which of them a given reducer changes.

Each step of Algorithm 2 anchors one of the six design choices along which the surveyed reducers differ, giving them a common conceptual map:

- **Program Representation**: how the input program P is mapped to a working representation P', such as a flat list of lines or tokens, a syntax tree, or a grammar-derived or domain-specific structure;
- **Transformation Space**: which transformation operators $\mathcal{T}$ are permitted under that representation, ranging from deletion alone to richer structural or semantic moves;
- **Search and Guidance**: how those transformation operators are instantiated into concrete candidates $\mathcal{K}$ and in what order the candidates are tested, whether by systematic traversal, search-prioritized heuristics, or learned relevance estimates;
- **State Update and Stopping**: what search state σ is maintained, how each oracle verdict updates P' and σ, and what condition terminates the search and yields the final result;
- **Oracle Engineering**: how each candidate is built, executed, and checked against the target property in practice—through construction of the *harness*, i.e., the infrastructure that builds, runs, and replays each candidate and invokes the oracle, together with result caching and parallelization—so that repeated oracle invocations stay efficient and reliable. The predicate ψ itself is fixed by the target failure; what reducers vary is how its value is obtained, e.g., a cached verdict is returned without evaluating ψ at all [86]. Algorithm 2 therefore models the invocation as its own step, test(p, ψ), which yields $\psi(p)$ under these engineering choices;
- **Validity Control**: what notion of validity each candidate must satisfy throughout the search, from parsability to type- or sort-correctness. Unlike the other five choices, it does not correspond to a single step of Algorithm 2. The choice is instead where along the pipeline validity is enforced, namely in the representation (Algorithm 2), in the transformation operators (Algorithm 2), or in a pre-check, such as a parse or type check, that rejects malformed candidates before the oracle runs (Algorithm 2). A reducer can also enforce it nowhere and leave validity to the oracle, though then some queries go to candidates that were never usable.

In *ddmin* (Algorithm 1), each of these choices takes a well-defined form: P' is a flat configuration; σ records the current partition granularity; its transformation space $\mathcal{T}$ is the single operator *delete*; generate_candidates instantiates that operator as the concrete subset and complement

removals at the current granularity, yielding $\mathcal{K}$. When a candidate preserves the target behavior, it becomes the new P'; when no candidate at a given granularity preserves it, σ is updated to increase the partition count; termination coincides with a local minimality condition under the current decomposition. Subsequent reducers extend this process by changing the representation, widening the transformation space, enriching the update logic, or strengthening validity and oracle handling.

The subsections below examine each design choice in turn; § 5 then reorganizes the same design space along four comparative dimensions, and § 6 details the transformation repertoire.

4.2 Program Representation

The first design choice concerns how the current program is represented, since the representation determines both which candidates can be generated and what validity guarantees the reducer can enforce. Classical list-based reducers such as *ddmin* treat the target program as a flat configuration[2] of elements, typically lines, characters, or tokens [31, 113]. This approach is broadly applicable: the reducer needs an oracle and a decomposition, but not a parser, grammar, or semantic model. When the target program contains strong structural dependencies, however, flat decomposition treats syntactically and semantically different elements as interchangeable units, often producing malformed intermediate variants and wasting oracle queries on irrelevant candidates [56, 78].

Subsequent reducers address this by making the representation more structured. HDD is the canonical first step, replacing flat partitions with tree-structured reduction over hierarchical inputs and substantially reducing the frequency of malformed intermediate candidates [56]. Reducers in the hierarchical and syntax-guided families of § 5 exploit a program's syntactic structure to guide their transformations. Perses takes a formal grammar as input and keeps every candidate parse-valid by construction [78], and Vulcan extends this syntax-guided line with richer moves [102, 104]. GTR is the exception, reaching such structure-guided moves without a grammar by generalizing reduction into tree-transformation templates and inferring plausible candidates from a corpus [30]. For solver inputs [6, 45], proof-assistant scripts [25], and other structured failure-inducing inputs, analogous domain models provide the reducer with representation-aware units that preserve the logical or syntactic invariants on which those inputs depend.

Across these designs, the representation ranges from a flat sequence with no structural model, through parse trees and grammars, to a domain model that encodes an input format's own invariants. The portability and setup costs along that range are discussed in § 5, where list-based versus hierarchical modeling and language generality versus specificity become explicit dimensions.

4.3 Transformation Space

Once a representation has been fixed, the next framework question concerns what candidate-generating transformations the reducer is permitted to apply. This choice matters because the available transformation space determines which reductions are reachable at all, not merely how efficiently the reducer finds them. Most transformations in the surveyed literature are *deletion-based*: every candidate they yield can be obtained from the current program by removing *elements* alone, where an element is a character, token, or other textual unit the program consists of. Deletion is the baseline throughout because it is broadly reusable and easy to justify against a failure-preserving oracle [113]. Deletion-based transformations differ mainly in how much structure guides the removal. The *ddmin* algorithm deletes a selected sublist of a flat list, or that sublist's complement, without consulting syntactic structure [113]. Perses instead deletes under grammar guidance, removing Kleene-quantified tree nodes or hoisting a node into its ancestor's place, a

[2]Throughout, *list-based* names this representation family and *flat* describes the configuration it operates on; both refer to the same non-hierarchical sequence of elements.

non-contiguous deletion of the enclosing elements [78], and GTR's templates delete a node or substitute it by one of its children [30]. A smaller set of transformations goes further and *rewrites*, substituting or introducing elements rather than only removing them, as in C-Reduce's renaming and inlining passes [66], Vulcan's identifier and subtree replacements [104], Latra's user-defined match-rewrite rules [105], T-Rec's lexical canonicalization [103], structure-form conversion, which rewrites a subtree into an alternative valid form of the same grammar nonterminal (detailed in § 6) [102], and LPR's model-proposed rewrites [114]. Transformation space thus determines the ceiling on what a reducer can achieve, independent of how its search is guided.

Enlarging that space is a consequential trade-off. Richer transformation spaces can produce smaller and more canonical failure-inducing programs, since moves such as identifier normalization and structure-form conversion clean up what deletion alone leaves behind. They also enlarge the search space and raise engineering cost. Whether candidates remain valid depends on how each transformation is designed rather than on how many the reducer has. Grammar-guided moves, and sort-preserving moves that respect the types of terms in solver inputs, keep candidates syntactically or sort-valid by construction, often more reliably than blind deletion [45, 78], whereas unconstrained rewrites can breach invariants that no parser checks [114].

4.4 Search and Guidance

Representation and transformations determine which reductions are reachable; prioritization and search policy determine how efficiently the reducer reaches a minimal one. The classical baseline is again *ddmin* partitioning: recursively test subsets and complements until the search reaches a local minimality condition [113]. That baseline remains influential because it provides a clear, reusable procedure that requires minimal domain-specific knowledge. Even reducers introduced mainly for their representation embed a default search order. HDD reduces level by level, breadth-first from the coarsest to the finest tree level [56], and Perses reduces node by node through a size-priority queue [78]. Later work makes this order itself the object of design, from heuristic prioritization in PARDIS [20], through probabilistic guidance in ProbDD and WDD [94, 120], to learned guidance that predicts which removals stay valid before testing them [21]. § 5 compares these strategies.

These contributions establish search as a first-class design choice, one that varies independently of representation and transformation choices. The central trade-off is between generality and signal quality: guidance that requires little domain-specific knowledge applies to any subject, whereas stronger ranking signals come from heuristics or learned models tailored to a particular setting. Better guidance can reduce wasted oracle queries dramatically, but only when the ranking signal, i.e., the score a reducer uses to order candidates before testing them, accurately predicts which candidates the oracle will accept. If that signal is weak, the reducer may abandon useful search directions too early or overfit to locally convenient candidates. Search-effort allocation is one of the main sources of variation across reducer categories, and claims about speedups depend heavily on how search improvements are measured.

4.5 State Update and Stopping

This design choice concerns how the reducer revises its state after each oracle verdict and when it declares the reduction complete; in Algorithm 2 these are the `on_success`, `on_failure`, and `should_stop` components. Update and stopping are not independent: the stopping criterion is the fixpoint of the update rule, reached when no further update makes progress under the current representation and interestingness predicate. For list-based reducers, this fixpoint is 1-minimality, where no single one-step deletion preserves the target behavior [24, 113]. For hierarchical reducers it becomes representation-specific, as in 1-tree-minimality, where no single tree node can be removed while preserving the failure [56].

Reducers differ mainly in how aggressively they revisit the search space before accepting that fixpoint: one-pass reduction accepts the first fixpoint it reaches [33], iterative variants re-run the reduction until a full pass yields no further change [89, 92], and post-pass refinements apply additional transformations to escape the local minima of ordinary 1-minimality [104]. A valid stopping criterion therefore does not settle whether the result is as small or as informative as a richer search model could achieve.

Stronger stopping criteria and repeated passes may produce smaller or cleaner failure-inducing programs, but they also increase oracle cost and complicate claims about what guarantee has actually been achieved. Iteration and stopping are therefore substantive design choices that determine how modern reducers balance oracle cost against output quality. This connects to the guarantee discussion in § 3, where the strength of a reducer's output guarantee depends on the stopping logic.

4.6 Oracle Engineering

Modern program reduction is also shaped by how the workflow makes oracle-driven search feasible and affordable in practice. This design choice does not mainly change which candidates are generated; instead, it governs how those candidates are actually executed, checked, cached, replayed, or parallelized under real oracle and environment constraints. These mechanisms include caching, parallelism, and replay support. For example, RCC treats caching as a reduction-specific design problem [86]. Picire parallelizes *ddmin*, running subset and complement tests concurrently without sacrificing 1-minimality [33], and GreeDDy stabilizes such parallel reduction by greedily merging the failure-inducing configurations that concurrent search can produce [93]. Execution-aware reduction uses record and replay to make difficult oracles usable [4]. Proof-assistant reduction and other environment-heavy settings further show that the oracle wrapper and its surrounding harness (build steps, dependency versions, and replay scripts) are often part of the reduction problem itself rather than merely part of evaluation [25].

The main trade-off here is between algorithmic simplicity and the caching, replay, and harness machinery that practical oracles demand. A paper can present a clean reduction algorithm in the abstract, yet its practical usefulness may still depend on caching, replay, harness engineering, or parallel execution. Treating these mechanisms as first-class workflow components makes explicit that oracle cost is a structural part of how reducers are designed and compared. It also creates a direct link to § 8.3.2, where the reporting of caching, replay support, and environment assumptions becomes one of the main determinants of reproducibility and fair comparison.

4.7 Validity Control

All of the preceding framework choices are bounded by validity. In this survey, *validity* is task-relative: a candidate is valid only with respect to the representation, language, oracle, and downstream use for which the reducer is being applied. For some reducers, validity is mainly syntactic: the program is required to remain parsable and structurally well-formed [2, 6, 56, 78]. For others, validity is semantic, behavioral, or domain-specific: the reduced program is required to remain compilable [28], logically meaningful [45], faithful to a proof or model [14, 25], or replayable in the environment needed to reproduce the target failure [4].

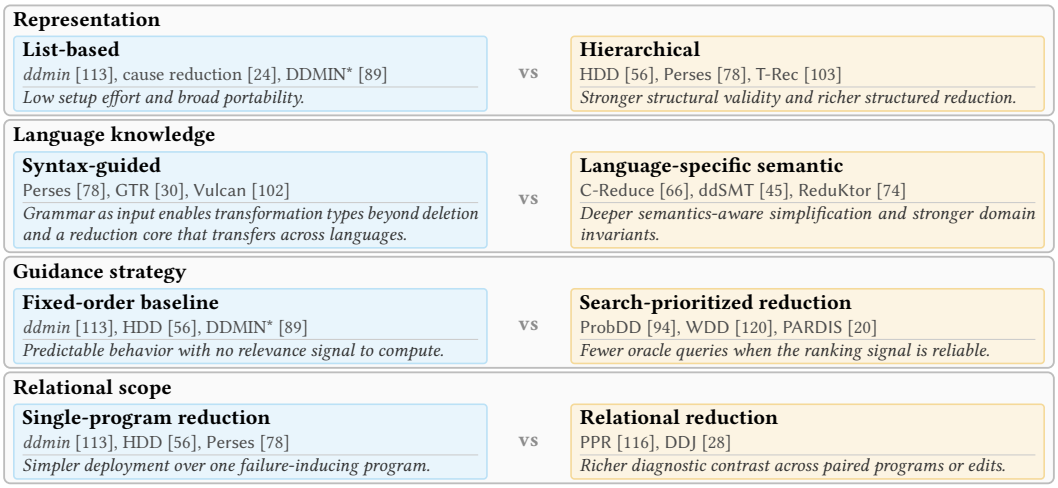
For that reason, validity control should be treated as a first-class design choice rather than as a post-hoc filter. Tree-based reducers [56], syntax-guided reducers [78], language-specific semantic reducers [28], and domain-specific systems [4, 14, 25] all encode different notions of validity, from parser-level well-formedness to proof, model, or replay validity. Beyond constraining the final result, validity shapes which representations are usable, which transformations are permitted, and which search directions are worth exploring.

Validity control, like the preceding design choices, involves a trade-off that is central to the rest of the survey. Stronger validity control usually narrows portability and raises engineering cost, but weak validity control can leave the reducer spending oracle queries on candidates that are cheap to generate yet fail for incidental, validity-related reasons rather than the target failure.

5 COMPARATIVE DIMENSIONS OF PROGRAM REDUCTION

Whereas § 4 decomposed reducers into procedural components, this section reorganizes the same design space along four comparative dimensions: representation [44, 113], language knowledge [104, 105], guidance strategy [94], and relational scope [112, 116]. Figure 3 pairs each with the reducers that sit at either pole. We single out these four because they are where reducers differ most sharply: the procedural choices co-vary in practice, with a reducer's representation largely fixing which transformations and validity guarantees are reachable, so a few dimensions capture most of what distinguishes one reducer from another. Note that: 1) the names of these dimensions are survey constructs rather than universal names from the primary literature; we adopt them because each exposes a recurring trade-off, in portability (how readily a reducer applies across languages and input formats), validity, search cost, and diagnostic value; 2) these dimensions are comparative rather than categorical, so a reducer can sit at more than one point along them; 3) transformation space cuts across representation, language knowledge, and guidance, so we discuss it procedurally in § 4 and mechanistically in § 6 rather than treat it as a fifth dimension.

Fig. 3. Comparative dimensions used in this survey to organize program-reduction techniques. Each card summarizes one dimension by showing its two poles, representative reducers, and the trade-off it exposes.



5.1 Representation

The representation dimension distinguishes reducers that treat programs as linear sequences from those that manipulate explicitly structured objects. List-based reducers operate over lines, characters, tokens, or other sequences [31, 60, 112, 113]. The canonical example is *ddmin*, which partitions and prunes those sequences without consulting syntax [59, 69, 113]. Later variants strengthen the search procedure or iterate beyond one local stopping point, but preserve the same list-based foundation [89, 92, 93].

Hierarchical reducers instead parse inputs into trees or related structured representations, reducing, though not eliminating, malformed intermediate candidates [32, 34, 44, 56]. HDD and its recursive variant HDDr reduce parse trees level by level [44, 56]. Syntax-guided reducers use structured representations differently: Perses keeps candidates parse-valid, i.e., accepted by the language's grammar [78], Vulcan adds more aggressive transformations [102, 104], T-Rec adds lexical canonicalization after structural reduction [103], and DDSet abstracts the reduced input into a generalized pattern over the same grammar-derived trees [22]. The trade-off is persistent, and neither end dominates: list-based representations give up structural precision for zero setup, while structured ones give up portability for candidates that are more likely to be well-formed.

5.2 Language Knowledge

The language-knowledge dimension asks how much knowledge of the target language a reducer's decisions depend on, ranging from none, through syntactic structure, to deep target-specific semantics. It applies to every reducer, not only to hierarchical ones, and list-based reducers such as *ddmin* sit at the zero end, treating the input as an opaque sequence [113]. At the low end the dimension co-varies with representation, since exploiting syntax presupposes at least a parser; the two separate at the semantic end, where reducers sharing the same tree representation differ sharply in how much semantics they encode, as Perses and DDJ illustrate [28, 78]. The primary literature names its two ends directly, contrasting language-agnostic with language-specific reducers [104, 105]: *language-agnostic* reducers reuse one reduction algorithm unchanged across languages, working over generic structure rather than hardcoded semantics [78], whereas *language-specific* reducers build in the semantics of a particular language or domain [28, 66]. Within the language-agnostic end, this survey uses the term *syntax-guided reducers* for frameworks that take a grammar as an explicit input to enable transformations that contiguous deletion cannot express, such as hoisting and structure-form conversion [78, 102, 104]. Because a grammar captures only syntax, however, these reducers can keep every candidate parse-valid yet still cannot guarantee deeper properties: a parse-valid C program may exhibit undefined behavior, and a parse-valid SMT formula may be ill-sorted, i.e., violate its theory's typing rules. Ruling out such candidates requires knowledge the grammar does not encode [66].

Language-specific semantic reducers close this gap by encoding rules about a particular language or domain. DDJ, for example, reduces Java source-code changes through AST-aware edit rules so that the remaining patch stays compilable and semantically interpretable [28]. C-Reduce pairs source-to-source passes with semantic validation, i.e., external checks that reject semantically invalid candidates, so that a reduced program's undefined behavior does not produce a misleading compiler-bug report [66]. ddSMT preserves the sorts and theory constraints that keep a reduced formula meaningful to the solver [45]. At the bytecode level, the same approach reduces failure-inducing Java bytecode while preserving reference dependencies and the method-overriding rules that keep a reduced classfile loadable [40, 41]. The barrier is expertise rather than effort: designing a sound rule demands deep knowledge of the target's semantics and the rules do not easily transfer to another language.

The dimension is therefore a continuum between portability and precision rather than a binary distinction between general-purpose and specialized tools. Latra sits in the middle: it reuses a unified transformation notation across languages, yet its rules still encode human-designed language knowledge [105]. LPR reaches the same middle ground from the opposite direction, drawing language knowledge from a pretrained large language model rather than hand-written rules, so no rules need to be written for each new language, but validity is left to the oracle [114]. A reducer buys deeper validity preservation with per-language engineering effort, so the right point on this continuum depends on how costly invalid candidates are in the target domain [3, 29, 43].

5.3 Guidance Strategy

The guidance dimension concerns whether a reducer prioritizes which candidates to test or simply follows a fixed order. In the baseline, that order is fixed by the representation and partitioning rather than by any estimate of relevance: *ddmin* tests partitions and complements in sequence, and HDD traverses the parse tree level by level [56, 113]. This keeps behavior predictable but blind to which candidates are most likely to be removable.

Search-prioritized reducers instead use a signal to reorder candidate evaluation. ProbDD estimates element relevance probabilistically [94, 95]; the WDD family refines partitioning through weights [120]; and PARDIS prioritizes AST nodes rather than following a fixed structural order [20]. The CDD analysis suggests that much of ProbDD's benefit stems from counter-like prioritization, i.e., deprioritizing elements whose removal has repeatedly failed, and from avoiding queries that are unlikely to succeed [115]. Such prioritization can sharply cut the number of oracle queries, but only when the ranking signal is reliable, and it does not expand the transformation space or relax validity constraints [43, 104]. The two ends therefore coexist by addressing different bottlenecks. The fixed-order baseline is simpler and suffices when the oracle is inexpensive. Prioritization, by contrast, repays its signal-computation cost only when oracle cost dominates, since its advantage is fewer expensive queries rather than smaller or more valid results.

Learning-based guidance sits at the boundary between the guidance and language-knowledge dimensions. One learned-guidance approach, for example, trains a classifier on syntactic features of the target language to predict which removals remain valid, so its guidance signal also encodes language-specific expectations [21]. Guidance of this kind, whether a trained classifier or a reinforcement-learned policy [29], ranks and filters candidates within the reducer's existing transformation space. Large language models can also supply ranking signals. In the surveyed corpus, however, their distinctive use is generating new candidate rewrites, which changes the transformation space rather than the guidance strategy and is therefore discussed with the transformation families in § 6 [114].

5.4 Relational Scope

The relational-scope dimension asks whether reduction targets one failure-inducing program or a contrast between two or more programs. Single-program reduction is more broadly deployable and suffices when the goal is a small, reproducible failure-inducing program [69, 113]. It falls short, however, when the failure is itself a *difference*. Suppose a fuzzer mutates a seed program that a compiler handles correctly, and the mutated variant crashes the compiler. Reducing the crashing variant alone yields a small program that still crashes the compiler, which shows what is needed to reproduce the failure. It does not show which of the mutations turned the seed into a failure-inducing program, because the reduced program retains no connection to the seed. Relational reduction targets that question directly. It minimizes the difference between an artifact that exhibits the failure and a reference that does not, leaving the smallest difference that still induces it.

This idea is already present in Delta Debugging's original formulation, which minimizes the difference between a failure-exhibiting run and a reference run [110, 112], whereas the *ddmin* algorithm minimizes a single failure-inducing input on its own, the special case with an empty reference (§ 3) [113]. Later work carries the relational view further. DDJ reduces the Java patch responsible for a regression rather than a whole program [28], and Pairwise Program Reduction (PPR) shrinks a failure-inducing variant and its reference together while preserving the failure-inducing difference between them [116]. The benefit is a sharper diagnostic, the minimal change that explains the failure, at the cost of a reference program that may not be available, higher setup, and a larger search that must keep both artifacts valid and their contrast intact [28, 102, 116].

Single-program reduction therefore remains the default, while relational reduction is worth its extra cost only when the contrast itself is the diagnostic goal.

6 THE TRANSFORMATION SPACE

The preceding sections treat the transformation space as one workflow design choice among several. This section examines it in its own right, because the transformation space sets a reducer's ceiling: it determines which candidate programs can be generated at all, which validity constraints hold by construction, and which local minima the search is likely to reach. We organize the surveyed transformations into four families by the amount of language knowledge they require, from deletion, which needs none, to semantic transformations, which encode domain-specific invariants; Figure 4 illustrates each family on one program. This grouping is independent of the deletion-versus-rewrite distinction of § 4: *deletion and pruning* only remove elements, *structural rewriting* always introduces new elements into the candidate, and the lexical and semantic families contain transformations of both kinds. The families are design choices, not a ranking; the portability and cost exchange that the ordering implies is set out in § 5.

```
int limit = 12345;
int check(int n) {
  if (n > 0) { return trigger(n); }
  return probe(n + limit);}
```

(a) Initial failure-inducing program

```
int limit = 12345;
int check(int n) {
  if (n > 0) { return trigger(n); }
  return probe(n + limit);}
```

(b) Deletion and pruning: chunk deletion and hoisting [56, 78, 113]

```
int limit = 12345;
int check(int n) {
  if (n > 0) { return trigger(n); }
  return probe(n + limit) 42 ;}
```

(c) Structural rewriting: subtree replacement [66, 102, 104]

```
int limit c = 12345 1 ;
int check(int n) {
  if (n > 0) { return trigger(n); }
  return probe(n + limit c);}
```

(d) Lexical canonicalization: in-token trimming and canonicalization [103]

```
int limit = 12345;
int check(int n) {
  if (n > 0) { return trigger(n); }
  return probe(n + limit 12345);}
```

(e) Semantic transformation: inlining a definition into its use [28, 66]

Fig. 4. An illustrative example of the four transformation families, each applied to the same initial program. Struck-through elements are removed relative to the initial program; shaded elements are newly introduced.

6.1 Deletion and Pruning

Deletion is the fundamental and most intuitive transformation of failure-inducing program reduction: since the goal is a smaller program, the direct move is to remove part of the current program and ask the oracle whether the behavior of interest remains [112, 113]. In list-based reducers such as *ddmin*, the removable units are usually characters, tokens, lines, or contiguous chunks [11, 24]. In structured reducers, the same idea appears as pruning over statements, productions, subtrees, clauses, or other representation-aware components [66, 78, 97]. Deletion is the most portable transformation family: it requires minimal language-specific knowledge and applies across any domain where candidates can be generated and tested.

Hierarchical and syntax-guided variants refine where deletion applies. HDD and its recursive variant HDDr prune parse trees level by level, discarding whole subtrees before descending into their children [44, 56], and GTR generalizes such tree edits into templates that delete a node or substitute it by one of its children, without requiring a grammar [30].

Deletion also extends beyond contiguous units. Hoisting replaces a subtree with a compatible descendant, as in Figure 4b, where the enclosing `if (n > 0) {` and `}` are deleted while the guarded `return trigger(n);` survives [78, 90]. Although implemented as a node replacement, its effect is

purely subtractive, since every token of the result already appears in the input in the same order; hoisting is thus a generalized, non-contiguous deletion, and the grammar's role is to identify which such removals keep the program well-formed.

Perses places such deletions on a syntax-guided footing. It first normalizes the target language's grammar into *Perses Normal Form* and parses the program into a syntax tree. It then traverses that tree with a worklist, choosing each move by the node's grammatical role: list nodes (grammar Kleene-star or -plus repetitions) are reduced by Delta Debugging, while ordinary rule nodes are replaced by a grammar-compatible sub-node, i.e., the hoisting above. Because every move stays inside the grammar, all candidates are parse-valid by construction. The search then iterates to a fixpoint at 1-tree-minimality [78].

Across these variants, from flat chunks to grammar-guided hoisting, the transformations remain purely deletion-based. Reducers that prioritize their search, PARDIS, ProbDD, and WDD, likewise keep exactly this deletion space and change only the order in which removals are attempted [20, 94, 120].

A fundamental limitation of deletion is its constrained search neighborhood, i.e., the set of candidates reachable from the current program in one transformation. A reducer may reach a state in which no single remaining unit can be removed even though the program still contains removable structure under a richer transformation space. Complement testing partly addresses this by keeping a selected partition and discarding the rest, but this more aggressive probe often generates malformed or meaningless candidates [42, 97]. For domains with strong syntactic or semantic constraints, unconstrained deletion can corrupt the object's semantics, as in probabilistic programs or solver inputs whose meaning depends on relationships among remaining components [14, 45]. Deletion therefore remains the reusable core of the field, but also the one whose limits most clearly motivate richer transformation spaces.

6.2 Structural Rewriting

Structural rewriting does what deletion cannot: it substitutes or introduces elements rather than only removing them, as in Figure 4c, where a call is replaced by the constant 42 that appears nowhere in the initial program. Operations that add elements may seem out of place in a task that aims to shrink the program, but rewriting earns its place by creating new reduction opportunities: restructuring can unblock deletions that were impossible in the original form, and substitution can simplify what deletion must leave intact. These transformations are usually enabled by parse trees, grammars, or other structured representations.

Vulcan, for example, builds on this unblocking idea by wrapping a reducer such as Perses in a refinement loop. Once deletion reaches 1-minimality, Vulcan applies rewrites (identifier replacement, subtree replacement, and local exhaustive enumeration over small subtrees, which tries every grammar-permitted form of a subtree) to restructure the program into a different but still failure-inducing form. It then feeds that form back through the reducer, so previously blocked deletions become possible. Iterating this loop lets the reducer move past the local minima that trap deletion-only reduction [104]. Structure-form conversion (SFC) takes a different route: it rewrites a subtree into an *alternative valid form* of the same grammar nonterminal. In practice, it may substitute a smaller equivalent structure, eliminate or merge identifiers, or canonicalize the shape. These moves unlock reductions that deletion, hoisting included, cannot reach, and yield more uniform, deduplication-friendly outputs [102].

Some reducers generalize this into a pipeline of many transformation passes. C-Reduce is the canonical example: it orchestrates a large suite of pluggable source-to-source passes, some deletion-like and others inlining, renaming, or otherwise refactoring C/C++ code, and applies them in repeated rounds until a full pass yields no further change [66]. Because such aggressive

rewriting can silently introduce undefined or unspecified behavior, each candidate is screened with external semantic checkers: a static analyzer (Frama-C) and a formal C-semantics interpreter (KCC). The reduced program therefore still demonstrates the bug with well-defined behavior, rather than behavior the standard leaves open [66]; ReduKtor applies a similar multi-stage pipeline for Kotlin [74]. More broadly, C-Reduce shows that reduction can be a modular, fixpoint-based orchestration of many composable transformations rather than a single minimization operator. This breadth has a cost: a larger and more expensive search. Each additional class of rewrite can unlock smaller or cleaner outputs, but it also increases the number of candidates to validate and the effort of keeping each intermediate program a well-formed failure-inducing input that later passes can continue to reduce.

6.3 Lexical Canonicalization

Lexical canonicalization operates below the granularity that tree-level reducers often treat as atomic. Instead of deleting statements or subtrees, it simplifies identifier names, literal contents, and other token-internal structure. T-Rec is the primary example: after ordinary syntax-guided reduction reaches a structural fixpoint, it applies localized token-level simplification, both trimming characters inside tokens and replacing tokens with simpler forms, so that residual identifiers and literals become shorter and more canonical, as in Figure 4d, which trims `12345` to `1` and renames `limit` to `c` [103]. Structure-form conversion touches the same granularity through its identifier-elimination step, though its main focus remains structural [102]. Such transformations matter because many reduced programs stay cluttered even after their syntactic structure is minimized.

The benefit extends beyond size reduction. Lexical canonicalization can make reduced programs easier to inspect and compare, which connects reduction to deduplication and other downstream workflows. The primary trade-off is brittleness: once a reducer edits token internals, the candidate space grows quickly and the preserved behavior may depend on details that appear lexically incidental, such as literal ranges, identifier relationships, or prefixes used by the harness. Practical lexical reducers therefore need attempt limits and validation checks to avoid spending excessive oracle queries on small but invalid or behavior-destroying mutations [103]. Compared with the other transformation families, however, lexical canonicalization is still underexplored: T-Rec is essentially the only dedicated technique, and most reducers continue to treat tokens as atomic.

6.4 Semantic Transformation

Semantic transformations are the most specialized part of the transformation space. They encode knowledge a grammar cannot, to preserve properties that syntax alone cannot express: a program's types and behavior, or the logical content of a solver formula or proof script. These properties run along a rough spectrum from static to dynamic. At the static end is well-formedness beyond parsing, such as type correctness for source code [28, 66] and well-sortedness for solver inputs [45], so that inlining a definition, simplifying a type-level computation, or rewriting a formula still leaves a program that type-checks or stays within its theory. In the middle lie logical obligations: the scopes, bindings, and proof goals that keep a reduced proof-assistant script replayable [25]. At the dynamic end is runtime behavior, the execution trace or the probabilistic-model semantics that the reduced program must still reproduce [4, 14]. Across this spectrum the common move is the same: replace blind deletion with edits that respect the relevant invariant, as in Figure 4e, where a definition is inlined into its use and only then removed.

What makes this knowledge worth encoding is that a syntactically valid reduction can still be semantically wrong, and such a result can actively mislead. A reduced C program that happens to exhibit undefined behavior, for example, may not be evidence of a compiler bug, so a syntax-only reducer can produce convincing but false reports. C-Reduce names this the *test-case validity*

problem and secures validity with external semantic checks that reject offending variants, rather than building it into its transformations [66]. Other reducers take the opposite route and encode the target's semantics in the transformations themselves. DDJ decomposes a Java patch into AST-level change components and groups them with rules that capture the semantic dependencies among changes, some drawing on call and control-flow graphs, so a candidate applies a change only together with the changes it depends on, keeping candidate patches largely compilable [28]; ddSMT rewrites solver inputs with sort- and theory-preserving transformations [45]; and RR-Reduce carries the idea to dynamic behavior, isolating the bug-triggering function of a WebAssembly binary and using selective record and replay to stand in for the code it removes, so deletion still reproduces the recorded execution [4].

These guarantees carry a price that is not only labor but expertise. Devising a sound transformation demands deep understanding of the target's semantics (its type system, theory, proof discipline, or execution model), since one built on a shaky grasp of them reintroduces exactly the invalid candidates that semantic awareness is meant to rule out. These transformations are therefore the strongest in validity yet the weakest in portability: each is bound to one language, hard to devise without domain expertise, and costly to maintain as that target evolves.

That double cost, domain expertise as well as per-target engineering, is what a separate line of work tries to escape. These approaches reach beyond syntax without actually encoding or verifying semantics, seeking language-aware transformation more cheaply than dedicated reducers allow. Latra applies reduction rules written in a small template language: each rule encodes a language-specific rewrite but is matched against the parse tree, which lets the rules be reused across languages [105]. Operating only at this syntactic level, the rules stop short of transformations that need genuine semantic information, such as pointer or runtime analysis. Large language model-aided reduction arrives at the same point by a different route: LPR, the first reducer to bring a language model into the workflow, has the model propose candidate rewrites that draw on knowledge absorbed during training and relies on existing reducers and the oracle to reject invalid or unhelpful ones [114]. Both obtain transformations that are more language-aware than syntax-only rewriting yet weaker than encoded semantic invariants, and both defer validity to the oracle rather than guaranteeing it in advance, but differ in where the knowledge lives: Latra externalizes it as human-written templates, whereas LPR draws on a model's implicit, unverified knowledge, trading transformation engineering for cost in proposal filtering and reproducibility.

7 APPLICATION DOMAINS

Program reduction began in automated debugging, but its machinery travels: this section surveys the domains that have adopted it and what each asks a reducer to preserve. We split the representative domains into two core ones and three adjacent ones. The two core domains, automated debugging and compiler and toolchain testing, apply the reduction workflow of § 4 to a failure-inducing program, shrinking it while an oracle checks that the failure still reproduces, and they are where the corpus concentrates. The three adjacent domains keep the same workflow but depart from the core in what they reduce, what they preserve, or both: trace and systems reduction [7, 27], AI-model understanding [63, 79], and software debloating [1, 54]. These transfers show how far the minimization idea extends when the object goes beyond a single failure-inducing program.

Table 1 pairs each domain with the object it reduces, the property it preserves, and the objective that follows. To gauge how these domains have grown, we count, for each time period, the papers in the reviewed corpus that match each application subject, and Figure 5 plots the result. Across five periods from 1999–2004 to 2020–2026, automated debugging remains the largest strand throughout, while compiler and toolchain testing rises sharply in the two most recent periods. Debloating and specialization grows after 2015, and AI-model understanding appears only in 2020–2026. Since

Table 1. Application domains of program reduction with representative papers. The two core domains preserve a reproducible failure, whereas the three adjacent transfers preserve other properties.

	Application domain	Reduced object	Preserved property	Objective
Core	Automated debugging and troubleshooting [8, 24, 113, 116]	Failure-inducing program or input	The reproducible failure	A small, inspectable reduced program whose causal structure eases root-cause analysis
	Compiler and toolchain testing [4, 25, 45, 66]	Failure-inducing test program or input	The tool's failure that the program triggers	A small, valid, human-readable bug report, distinguishable from reducer artifacts
Adjacent	Trace, execution, and systems reduction [7, 27, 37, 97]	Execution trace, configuration	Replay fidelity of the behavior	A smaller replay or explanation artifact, free of irrelevant dynamic detail
	AI model understanding and explainability [64, 79–81]	Model input, often a code fragment	The model's prediction	A faithful minimal fragment that reveals the evidence the model relies on
	Software debloating and specialization [1, 55, 82, 99]	Deployed binary, bytecode, or library	A required functionality envelope	A leaner deployable artifact that keeps required behavior, with size secondary

nearly every reduction paper mentions debugging, a paper that fits a more specific subject, such as compiler and toolchain testing, is counted only under that subject, not under automated debugging. The counts therefore indicate trends rather than an exhaustive classification. Trace and systems reduction, which lacks data for a stable trend, is omitted from Figure 5 and appears only in Table 1.

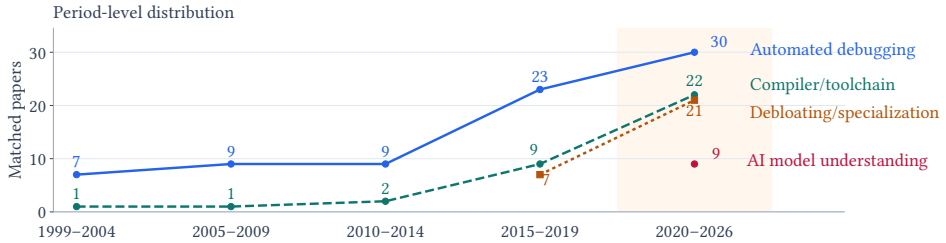


Fig. 5. Application-subject trends in the reviewed corpus.

7.1 Automated Debugging and Troubleshooting

Automated debugging is the founding application of program reduction and remains the largest strand in the corpus throughout the period surveyed (Figure 5). Reduction here starts from a failure-inducing program or input too large to inspect directly and shrinks it while keeping the failure reproducible [8, 111, 113]. What sets the domain apart is its consumer: a human developer performing root-cause analysis, for whom size is only a means. The objective is a result that is both small and still shows why the program fails, so the domain's techniques can be read as successive layers of diagnostic support added on top of bare minimization.

That progression runs from isolating the failing fragment to pinpointing the faulty code. The simplest approach runs *ddmin* directly on a single failing program or input, shrinking it until what remains is small enough for a developer to read by hand while still triggering the fault [113]. PPR reduces the bug-triggering program in tandem with a paired seed program, keeping both in sync so that what survives is the minimal difference between them, namely the code change responsible for the bug rather than merely a small program [116]. Dynamic-slice intersection adds fault localization on top of reduction. Once Delta Debugging has isolated the minimal failing input, the technique intersects a forward dynamic slice from that input with a backward dynamic slice from the faulty output, narrowing the result to the statements that lie on a path from the failure-inducing input

to the erroneous output [26]. Bouillon et al. deliver this inside the developer's environment as an Eclipse plug-in that runs Delta Debugging over JUnit tests and CVS history, so isolating a failure-inducing change happens in the IDE rather than as a separate step [8].

The distinctive cost in this domain is the loss of diagnostic context. Because a person reads the result, the smallest program is not always the most useful: the surrounding declarations, the data and control flow that lead to the fault, and the statements that show why it occurs are exactly what a developer needs to understand the failure, and aggressive deletion removes them [116]. The stopping point therefore rests on human judgment rather than a formal minimality criterion, which is why the techniques above preserve or recover the faulty code's context instead of optimizing size alone. Debugging therefore prioritizes practical utility, in the form of diagnostic value, over effectiveness (§ 3.3).

7.2 Compiler and Toolchain Testing

Compiler and toolchain testing is the most active application of program reduction in the recent literature. It differs from general debugging in where the fault lies: the program under reduction is only the trigger, while the defect resides in the compiler, solver, or verifier that processes it. Large generated tests [66], solver inputs [10], proof-assistant scripts [25], binaries, and bytecode programs [40] are reduced into compact failure-inducing programs for triage and root-cause analysis [38]. C-Reduce established that such reductions should be small, valid, human-readable, and directly usable in bug reports [66]. The demand is explicit in practice: both GCC and LLVM ask bug reporters to submit reduced failure-inducing programs [18, 52], which makes reduction a routine step in compiler bug triage rather than an optional convenience. The same triage role extends to optimization solvers: MIP-DD, the MIP counterpart to ddSMT, reduces a mixed-integer programming (MIP) instance to a few variables and constraints while preserving the solver's faulty behavior, and its reductions supported 24 of 51 documented bug fixes across several SCIP releases [35]. Reduction also narrows what a bug report discloses when the original program contains sensitive data. Because a reporter submits the reduced program rather than the original, and the reduced program retains only what is needed to trigger the fault, less of the original artifact reaches the developer. This can make a failing case shareable across an organizational boundary: a reduced and renamed MIP instance, for instance, exposes little of the original model's proprietary data, though how fully reduction anonymizes a case depends on the artifact, since reduced source code can still reveal algorithmic intent [35].

The central difficulty here is validity. Because the fault is in the tool, a reduced program is evidence only if it remains well-defined: aggressive deletion can introduce undefined or unspecified behavior and turn a genuine compiler bug into a false positive, the test-case validity problem [66, 106]. C-Reduce addresses this by folding semantics-based validation, with checkers such as KCC and Frama-C, into the interestingness test, so candidates with undefined behavior are rejected during reduction rather than reported afterward [66]. Program reconditioning attacks the same problem upstream, rewriting tests to avoid undefined behavior so a reduced bug stays valid across compilers rather than only the one that found it [48]. Porting a reducer to a new target raises the same concern: extending C-Reduce to OpenCL kernels required adding accurate detection of uninitialised-data accesses so that reduction does not convert a genuine compiler bug into a false positive [61].

These constraints favor syntax-guided, semantic, and execution-aware methods. List-based reduction remains a useful baseline, but systems such as C-Reduce [66], Perses [78], Latra [105], ddSMT [45], typed-language reducers [68], proof-assistant reducers [25], and WebAssembly-oriented systems like RR-Reduce [4] show why structure and oracle integration matter for toolchains. Validity is the dominant concern here (§ 3.3): it is the primary form practical utility takes for a compiler bug. In return, the domain accepts higher oracle cost, lower portability, and the deep

language-specific knowledge (§ 5) that preserving it demands. Because compiler fuzzing produces many failure-inducing programs that map to the same underlying defect, reduced reports are deduplicated to distinct bugs, rewarding a canonical reduced form: equivalent failures that converge on a comparable representation collapse to a single report. T-Rec drives equivalent reduced programs toward such a form through lexical-syntax-guided passes [103], an open challenge in § 8.

7.3 Trace, Execution, and Systems Reduction

Trace, execution, and systems reduction retains the form of a minimization problem but changes the object being reduced. The target may be an event trace, API-call recording, execution schedule, distributed interaction history, or system configuration whose simplification exposes the dynamic circumstances of a failure [7, 27, 37, 97]. The goal is a smaller explanation or replay artifact rather than a smaller failure-inducing program, so practical utility dominates: the result is evaluated by replay fidelity (a form of validity) and explanatory value (a form of diagnostic value). A reduced trace should still trigger, replay, or localize the behavior while removing irrelevant events, nondeterministic noise, or cross-component clutter.

Three properties make this harder than reducing static source, and each strains a different design choice from § 4. First, the artifact is replayed against a live, often nondeterministic system, so the same reduced trace may reproduce the failure on one run and not the next, which breaks the consistency that *ddmin* assumes and, applied directly, confines reduction to short sequences [37]. Second, events are bound by ordering and def-use dependencies, where a later event uses state an earlier one set up, so an event cannot be deleted as freely as an independent syntax subtree, because removing a setup event can invalidate every later event that depends on it [97]. Third, validity here means remaining replayable, so the oracle is dynamic and tied to record-replay infrastructure, and the reduced trace is often not portable outside the harness that produced it [7, 27].

The adaptations follow from these constraints. Nondeterministic reduction pairs a deterministic replay mechanism with event abstraction, grouping low-level events into stable units before searching for a minimal failure-inducing trace [37]. Dependency-aware reduction encodes ordering and type constraints directly, as in context-based event-trace reduction that models each event's variable and DOM usage as a constraint-satisfaction problem, so subtraces that could not replay are never generated [97]. Where a faithful replay harness already exists, *ddmin* transfers with little change once the right element granularity is chosen, as in OpenGL API-call traces and recorded web sessions [7, 27]. Trace and systems reduction therefore marks the boundary of the survey's scope: it shows how quickly reduction changes once the preserved object becomes an execution history rather than source code or another failure-inducing program.

7.4 AI Model Understanding and Explainability

In AI model understanding, the reduced object is a model's input, often a program or code fragment but sometimes other text, while the preserved property is a model's prediction rather than a reproducible failure. Reducing the fragment until the prediction is about to change exposes which features the model actually relies on, such as bug-bearing code, identifier names, method signatures, or other structural cues [17, 63–65, 71, 79]. This is a recent adjacent line of work in the corpus as neural code intelligence became a downstream consumer of program reducers.

Two differences separate this from reducing a failure-inducing program, and both reshape the oracle design choice of § 4. The oracle is a learned, black-box function rather than a deterministic check, so reduction preserves a statistical prediction and is best read as extractive interpretation, i.e., an explanation extracted from the model's observed behavior, rather than proof [63]. This creates a failure mode with no analogue in classical reduction: a minimal fragment can keep the prediction for the wrong reason, so the goal is not the smallest input but an audit of whether the

model attends to genuine signal [79]. Validity is contested too, because requiring the fragment to stay compilable bounds how far reduction can go, whereas dropping that requirement reduces more aggressively but may yield fragments that fail to compile [71, 79].

The adaptations center on the oracle and the element granularity. Because the reducer queries the model only as a black box, the oracle is simply whether the model still emits the same prediction, which lets Delta Debugging apply directly. SIVAND searches for the minimal prediction-preserving fragment of an input [63], and prediction-preserving input minimization reduces a sample to a 1-minimal, valid form to probe a detector's signal-awareness, i.e., whether its prediction rests on task-relevant code rather than incidental cues [79]. Tree-level variants such as SSGPS instead prune the abstract syntax tree using attention-derived node importance, so the kept fragment stays meaningful to the model [71, 81]. A related thread reuses the same machinery for efficiency rather than explanation. DietCode prunes tokens and statements by attention weight to cut inference cost while preserving model performance [118]. Across these uses practical utility, in the form of a faithful explanation, outweighs effectiveness, since aggressive simplification can destroy or distort the evidence being studied [63, 79].

7.5 Software Debloating and Specialization

Software debloating and specialization reduce deployable binaries, bytecode, libraries, or feature sets whose unused components create security, maintenance, or performance overhead [1, 9, 54, 55, 73]. The goal is to preserve a functionality envelope while removing dormant or unnecessary behavior, with output size a secondary concern.

Debloating inverts the usual reduction problem. There is no single failure to preserve. The property is a whole functionality envelope, and the oracle is whether the program still serves its required use cases rather than whether it still triggers a fault. That oracle can only be approximated, by a test suite, a usage profile, or static reachability analysis, all of which are necessarily incomplete [1, 54]. The dominant risk is therefore the opposite of validity loss in core reduction: removing code that no benchmark exercises but deployment still needs, a regression that surfaces only later [9, 54]. Dynamic behavior compounds this, since reflection, dynamic loading, and address-taken functions defeat static reachability and make it hard to establish safely which code is unused [1, 54].

In the terms of § 5, debloating sits at the language-specific end of the language-knowledge dimension, so the adaptations replace oracle-guided trial deletion with dependency- and reachability-aware analysis. Nibbler reconstructs a function-call graph from binaries and conservatively prunes only provably unreachable code [1]. XDebloat drives feature-oriented removal through a type-safe program-dependence-graph analysis [82]. JSHRINK combines static reachability with dynamic profiling to keep behavior intact under Java's reflective features [54, 99]. Because soundness outweighs aggressiveness, these systems often accept smaller reductions in exchange for safety, making debloating the domain where effectiveness and practical utility (§ 3.3) diverge most sharply.

8 OPEN CHALLENGES AND FUTURE DIRECTIONS

The preceding sections present program reduction as a broad and active design space, yet several of its most consequential problems remain open, and they lie beyond the reduction algorithm itself. They are questions of guarantee, cost, and credibility: what a reducer should preserve, how affordably and generally it can reach a result, and whether that result stays meaningful outside the setup that produced it. This section organizes the open challenges along those lines.

8.1 Canonicality, Validity, and the Preserved Property

The first group concerns the output a reducer delivers: its canonical form, validity, and preserved property. Each is a guarantee the result must carry, independent of how the reduction was reached.

8.1.1 Canonicalization. Canonicalization asks reduction to produce a stable, comparable representative, so that two reductions of the same bug converge on the same form. It matters most for deduplication and triage, which collapse reports to one root cause only when their reduced forms line up [103].

The difficulty is that minimality and canonicality are orthogonal goals, and the workflow of § 4 optimizes only the first. Deletion-based reducers strip irrelevant elements but leave the surviving form underdetermined: many distinct 1-minimal programs can trigger the same bug, and which one a reducer returns depends on traversal order, oracle timing, and starting point rather than on any preferred representative. Richer transformation spaces only enlarge the set of reachable forms, so without an explicit criterion for preferring one, they make outputs less reproducible [102, 114].

Current work shows canonicalization is attainable, but only in narrow, purpose-specific forms. T-Rec canonicalizes through lexical-syntax-guided passes that drive equivalent programs toward a common form for deduplication [103], and one line derives reduction and deduplication together from a shared set of semantics-preserving transformations [13]. Both fix a particular normal form for a particular purpose. Neither offers a general, representation-aware definition of what canonical should mean, or a way to certify that two reducers would agree on it.

Because the right normal form depends on the task, the open problem is to make the choice of form explicit and to decide where it is enforced. The form must be defined per task: a deduplication-oriented form collapses superficial variation, while a diagnosis-oriented form retains explanatory context, each fixed by a stated equivalence relation. Enforcement can then happen at either of two points. It can run after reduction, as a normalization pass that is *confluent*, i.e., whose result does not depend on the order of rewrites, so any reducer's output is standardized post hoc. Or it can be folded into the search itself, as a canonicality objective that accepts a transformation only when it moves toward the preferred form. Either way, the property becomes measurable, since testing whether independent reductions of the same bug converge turns canonicality from an assertion into a metric and makes reduced artifacts comparable across tools and runs.

8.1.2 Representation-Relative Validity. Validity asks if a reduced candidate remains structurally and semantically intact enough to use, and the bar is relative to purpose. It may require parsable source, a replayable execution, a well-sorted formula, or a program that still supports the debugging task it was reduced for. A result that is smaller but invalid for its purpose is unusable, which makes validity, not size, the binding constraint on how much a reducer can usefully shrink the program.

Validity failures arise when the transformations a reducer applies break the structural and semantic invariants the program must satisfy [78, 102, 103, 112]. Which transformations are safe is representation-specific, and the stronger the notion of validity, the more per-language domain knowledge it takes to tell safe transformations from unsafe ones, the knowledge that the language-specific semantic reducers of § 5 demand. The effect on effectiveness cuts both ways: restricting a reducer to valid candidates concentrates effort on viable variants and can reach smaller results for a fixed number of oracle queries, yet forbidding unsafe transformations also puts some reductions out of reach. The real trade-off is therefore generality, because stronger validity guarantees rely on grammars, parsers, or domain models that are representation-specific and costly to build and maintain. Achieving strong validity without that per-domain cost is the open problem the rest of this subsection examines.

Two strategies bracket current practice, each with an inherent limitation. Generate-and-test lets the reducer propose freely and rejects invalid candidates at the oracle, which is general but spends queries on candidates that were never going to be valid. Construct-valid systems embed the invariants directly: DelBugV does so for neural-network verification queries, program reconditioning for undefined-behavior-free compiler tests [3, 15, 48], and MIP-DD for optimization

instances, where every modification must preserve a reference solution's feasibility [35]. This strategy is efficient but applies only under narrower assumptions and demands per-class expertise.

The productive direction lies between these two extremes, building validity into the search rather than applying it as a filter afterward. Validity-aware operators produce only well-formed candidates by construction; dependency-aware search respects def-use and typing invariants without hard-coding a normal form; cheap intermediate checks prune doomed candidates before the expensive oracle runs; and guidance that exploits the target representation's constraints steers the search toward viable candidates [80]. What these share is a need for domain-calibrated guarantees that state which notion of validity is preserved, for which class of subject programs (e.g., a language or benchmark family), and at what cost in oracle queries, search flexibility, and portability [92, 101]. Validity decides whether a reduced result can be used at all, so it belongs in a reducer's design.

8.1.3 Generalizing the Preserved Property. Reduction shrinks an artifact while preserving some property of it, and in practice that property has almost always been a reproducible failure. Nothing in the reduction machinery, however, requires the property to be a failure. It needs only an oracle that decides whether each candidate still has the property. § 7 already showed the same machinery preserving replay fidelity, a model's prediction, or a functionality envelope by changing only what the oracle checks, and the reducers of § 5 are largely indifferent to which property that oracle encodes. The natural step is to treat the preserved property itself as the variable: reduction as property-preserving minimization, with failure reproduction as one instance among many. Generalizing that property beyond a binary failure signal, to graded, quantitative, or multi-objective targets, reaches past this survey's scope.

Current work has taken the easy half of this generalization. Cause reduction broadens the preserved effect to a user-defined one such as code coverage, reducing passing tests into fast regression suites rather than bugs [24], and the prediction-preserving reducers of § 7 do the same for a model's output label. In each case, though, the oracle stays binary. The field has generalized *what* is preserved while leaving *how* tied to a pass/fail check. The open problems begin where the property becomes graded, quantitative, or multi-objective.

The obstacle is the binary oracle itself. The *ddmin* algorithm and its descendants assume an interestingness test that returns yes or no, and both the subset-and-complement search and the 1-minimality guarantee are defined against that binary signal (§ 3). A graded property breaks this: when interestingness becomes a coverage percentage, a latency figure, or a model's confidence, the experimenter must choose a threshold, the property of a combined candidate no longer follows from the property of its parts, and minimality becomes ill-defined. Multi-objective targets compound the problem, replacing a single minimum with a Pareto frontier of incomparable forms, none of which is smaller on every objective. The difficulty is already visible in practice: a recent differential-testing study of coverage tools found automatic reduction straightforward for crashes, whose interestingness test simply parses the log for an assertion message, but much harder for coverage inconsistencies, where no comparably clean invariant exists [117].

Addressing these cases may require new minimality notions for non-binary properties. Quantitative targets call for a tolerance-bounded (ϵ-)minimality, i.e., minimality among candidates whose property value stays within ϵ of the original's [5], and multi-objective ones for a Pareto-minimality [107], each paired with a search procedure that replaces *ddmin*'s pass/fail recursion once the signal is graded. Framing reduction as property-preserving minimization then carries a clear benefit: an advance for one property class transfers to the others.

8.2 Cost, Guidance, and Generality

The second group concerns the search that produces the result: its cost, its guidance, and its generality across languages.

8.2.1 Scalability Under Expensive Oracles. Scalability in reduction is governed less by the size of the program than by the cost of the oracle. Every candidate must be checked, and when that check is a compiler invocation, a solver call, a replay against a live system, or an expensive test workflow, it is the number of oracle queries, not the number of edits, that sets the running time.

This makes oracle-query count the central quantity, but two complications keep it from being a clean target. First, queries are not equal: one that recompiles a large candidate costs far more than one rejected early, so minimizing query count and minimizing wall-clock time can diverge. Second, the cost is usually external and opaque to the reducer, living inside a build system or harness it cannot model, so a strategy that is efficient against one oracle is wasteful against another. Reported speedups are therefore conditional on program structure, oracle behavior, and infrastructure in ways easily mistaken for general algorithmic gains.

Existing work lowers oracle cost in three ways. Probabilistic and weighted prioritization try likely-removable elements first, improving *ddmin*'s worst case, though the gains depend on how well the model fits the program [94, 115, 120]. Caching avoids re-asking the oracle about equivalent candidates, though current schemes assume deletion-based reduction and cannot yet handle rewrites [86]. Parallel and hoisting-based search spread or shortcut the work, with gains bounded by dependence structure and coordination overhead [83, 91, 93]. None removes the need to query the oracle; each only makes the queries fewer, cheaper, or more parallel.

Because these mechanisms are complementary, the goal is to combine and characterize them rather than crown one winner. Two ideas target the cost directly: oracle-cost models that predict a query's price let a reducer prioritize on expected cost rather than raw count, and caching that works across runs, across programs, and across transformations amortizes oracle work beyond a single reduction. A third lowers cost by approximation, using *proxy oracles*, cheap stand-ins such as a fast static pre-filter or a learned predictor, to screen candidates before the true oracle runs, trading accuracy for throughput. The harder need is methodological: an evaluation discipline that records query counts, per-query cost, caching and parallelism assumptions, and where the bottleneck lies, so an environment-specific speedup is not mistaken for a methodological one. Oracle cost remains the main barrier to scaling reduction to whole projects, expensive verifiers, and interactive use.

8.2.2 Learning-Guided and LLM-Aided Reduction. Learned guidance and large language models are entering reduction as a way to propose transformations and prioritize candidates beyond those available to syntax-guided reducers. This extends the guidance dimension of § 5 and pushes the transformation space of § 6 past fixed syntax-guided rewrites toward open-ended, semantics-changing edits. The shift is in oracle economics. A classical reducer pairs cheap, deterministic edits with an oracle that is the expensive part. A model-driven reducer adds a second expensive, and now nondeterministic, component on the proposal side. A generated rewrite can also change meaning silently, so it cannot be trusted the way a deletion can. Because the model is part of the method, a result depends on the model version, prompt, and decoding parameters, which drift over time. These are sharper forms of two concerns that recur throughout this section: results that hold only relative to the surrounding pipeline, and reporting that omits what reproduction requires (§ 8.3.2).

Current work is hybrid. LPR first shrinks a program with a language-generic syntax-directed reducer to fit the model's context window, then applies LLM-proposed semantic transformations to go further than deletion alone [114]. This helps precisely where § 6 runs out of syntax-guided

transformations, but LPR also shows the limits: each model edit must still be re-checked by the oracle, model queries dominate cost, and the design space of how to use the model is barely explored.

How best to place the model in the loop is itself the open question, and its role can take several forms. It can propose transformations, with every rewrite gated by the property oracle, as in LPR. It can instead rank candidates while a classical reducer does the removing, keeping the expensive component off the critical path. It can act as a fast proxy oracle that screens candidates before the true oracle runs. Or it can be reserved for the residual that cheaper reducers cannot simplify. Whichever role it plays, the same hygiene makes it credible: report model version, prompt, and seeds, account for query cost against a documented limit, and gate every semantic rewrite with a validity re-check. Used this way, learned rewriting could bring semantic reduction to languages that have no hand-built semantic reducer, which connects directly to the domain-expertise barrier below, provided cost and reproducibility are controlled.

8.2.3 Lowering the Domain-Expertise Barrier. Semantic, validity-preserving reduction sits at the most powerful end of the language-knowledge dimension of § 5, and it is also the most expensive to reach, because it demands deep, per-language knowledge of the invariants that keep a reduced program meaningful. This is why the strongest reducers cluster around a few well-resourced languages and rarely transfer to a new one without a from-scratch effort.

The expense is really two costs that are easy to conflate. One is engineering: integrating with a compiler front-end, a parser, or a toolchain. The other is expertise: knowing which rewrites preserve meaning in the target language. Tooling can lower the first, but the second is irreducibly about understanding the language, and it decides whether semantic reduction reaches a new domain.

Latra addresses the engineering half. Its template-based, language-agnostic framework uses a lightweight rule language and parse-tree pattern matching to decouple transformation logic from compiler-toolchain infrastructure, so adding a language does not require rebuilding tooling [105]. What it does not remove is the expertise half: the rules themselves still encode human-authored, language-specific knowledge of what may be rewritten safely. Latra makes that knowledge easier to apply, not easier to acquire, which is exactly where the open problem now sits.

Lowering the expertise cost means trading soundness against effort, and the options span that spectrum. Curated libraries of transformation rules, shared and reused across related languages, are sound but labor-intensive to build. Mining candidate rules automatically from grammars or compiler front-ends scales better but yields rules that may be unsound and need validation. Learning a language's validity constraints from corpora, or having a language model draft transformation rules, could bootstrap a new language cheaply, which ties this challenge back to learning-guided reduction above, at the price of having to verify what is learned. Transfer across related languages, reusing the rules of one to seed another, helps where languages share structure. A controlled study of authoring effort, which Latra itself calls for, would for the first time measure how large the expertise barrier really is. Success here would let the most effective reducers serve the many languages that today have none, instead of the few that can fund a from-scratch expert effort.

8.3 Robustness and Comparability

The third group concerns what happens to a result afterward: whether it survives a change of context, and whether it can be compared with results from other work.

8.3.1 Robustness Across Contexts. Robustness is a question of artifact portability. A reduced program is meaningful only relative to the pipeline that produced it. Move it to a different compiler version, oracle wrapper, build configuration, or replay environment, and the guarantee can quietly evaporate [48, 114]. This concerns the artifact itself, not how results are reported: even a perfectly reported result can be fragile across contexts.

The reason is that a reduced program is minimal only with respect to one exact setup, and reduction removes the slack that would let it survive elsewhere. Stripping context can leave behavior that depends on a particular oracle timeout, an uninitialized value, or undefined behavior that happens to manifest under the original compiler. Tightening toward minimality worsens fragility in three ways: the target property underspecifies which behavior must survive a change of context, the reduction passes through brittle variants, and the result is certified by a single oracle and one local minimality condition.

Existing answers buy robustness with specialization. Program reconditioning rewrites generated tests to avoid undefined behavior so reduced compiler bugs stay valid across compilers rather than only under the one that found them [48], and domain-specific reducers raise robustness for a particular program class [71, 80, 92]. Both work by narrowing the assumptions under which the result is meant to hold, which is effective but does not generalize on its own.

The way forward is to make robustness explicit and testable rather than incidental. A reducer can declare a robustness scope, naming what behavior it preserves and under which execution or validation contexts. The scope should also state how sensitive that guarantee is to a change of oracle, of environment, or of the program's representation, such as reformatting or re-parsing. It can reduce against several oracles or configurations at once, so the result is minimal with respect to a family of setups rather than one. It can build validity into its operators, as reconditioning does, so brittle states never arise. Finally, it can report a sensitivity analysis, showing how the reduced program degrades as the oracle or environment is perturbed. This matters for reuse: a robust artifact survives in a bug report, a regression suite, and across tool versions.

8.3.2 Comparable and Reproducible Evaluation. Unlike robustness, which concerns a single artifact, this challenge concerns the infrastructure for comparing and reporting results across papers. Comparing reducers fairly is itself an open problem, because reported results rarely rest on aligned conditions. A meaningful comparison needs three things to hold together: comparable program classes, compatible oracles and harnesses and stopping criteria, and outputs evaluated against the same criterion [46, 66, 78]. When any of these differs, the reported numbers no longer measure the same thing, even when the programs look alike. C-Reduce, Perses, and ddSMT therefore serve as common baselines only within communities that already share subjects and harnesses [45, 84].

Closing that gap needs stronger benchmark and harness infrastructure. A benchmark is not only a set of subject programs. In program reduction it also includes oracle wrappers, parser assumptions, replay scripts, compiler flags, instrumentation logic, caching behavior, and stopping conditions that determine which reductions are admissible [46, 57, 83, 88]. Without that surrounding harness, a nominally shared subject set does not guarantee a shared experiment.

Benchmark choice also forces a trade-off between comparability and ecological validity, i.e., how well benchmark subjects reflect real deployment conditions. Historical infrastructures such as SIR, Siemens subjects, and long-lived examples including *Grep* and *Space* support repeatable comparison across reducers and over time [88, 112], whereas bespoke bug reports, project histories, compiler fuzz tests, solver inputs, and WebAssembly engine benchmarks better reflect deployment conditions but are not interchangeable across papers [4, 25, 66, 83, 102, 105]. Neither style suffices alone: a standardized component supports cross-tool comparison while a domain-specific component supports deployment credibility, so studies should include both or justify why one is not applicable, since no benchmark covers all of these classes equally well.

Comparability finally depends on what each paper reports, and current practice reports unevenly: measurements are packaged differently, baselines are chosen inconsistently, and the surrounding setup is often left implicit [12, 19, 21, 33, 50, 62, 74, 87, 97]. At a minimum, a study should report the final size relative to the original and to a named baseline, the oracle-query count or wall-clock time

under a documented oracle, the validity constraints and how they were enforced, and the stopping criterion used, noting why any of these does not apply. The reducer should be released together with its harness scripts (including the oracle wrapper), subject provenance, and dependency versions, since each is needed to reproduce or reuse the reported numbers [25, 88]. These gaps yield to no single instrument: standardized suites, living benchmarks that track deployment, shared reporting standards, and artifact-evaluation tracks each address part of the problem, and a healthy subfield will likely need all of them.

9 CONCLUSION

In this survey, we studied program reduction as a field in its own right. We assembled a main corpus of 67 papers and proposed an abstract reduction framework that expresses a reducer as a small set of recurring design choices. We further compared reducers along those choices, such as the dimensions of representation and language knowledge, grouped the surveyed transformations into four families ordered by the language knowledge each requires, and reviewed the field's application domains and open challenges. We call on future research to continue along three lines: to make explicit what a reducer preserves and at what cost, to establish that such guarantees still hold outside the setup that produced them, and to report results that are comparable across studies.

REFERENCES

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM, San Juan Puerto Rico USA, 70–83. https://doi.org/10.1145/3359789.3359823

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools* (2 ed.). Addison-Wesley, Boston, MA. https://www.pearson.com/en-us/subject-catalog/p/Aho-Compilers-Principles-Techniques-and-Tools-2nd-Edition/P200000003472/9780133002140

[3] Muaz Ali, Rumaisa Habib, Ashish Gehani, Sazzadur Rahaman, and Zartash Uzmi. 2023. BLADE: Towards Scalable Source Code Debloating. In *2023 IEEE Secure Development Conference (SecDev)*. IEEE, Atlanta, GA, USA, 75–87. https://doi.org/10.1109/SecDev56634.2023.00022

[4] Doehyun Baek, Daniel Lehmann, Ben L. Titzer, Sukyoung Ryu, and Michael Pradel. 2025. Execution-Aware Program Reduction for WebAssembly via Record and Replay. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, Seoul, Korea, Republic of, 816–827. https://doi.org/10.1109/ASE63991.2025.00073

[5] Borja Balle, Prakash Panangaden, and Doina Precup. 2015. A Canonical Form for Weighted Automata and Applications to Approximate Minimization. In *Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 701–712.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2024. *The SMT-LIB Standard: Version 2.6*. Technical Report. SMT-LIB. https://smt-lib.org/papers/smt-lib-reference-v2.6-r2024-09-20.pdf

[7] Daniella Bársony. 2022. OpenGL API call trace reduction with the minimizing Delta debugging algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. ACM, Singapore Singapore, 53–56. https://doi.org/10.1145/3548659.3561308

[8] Philipp Bouillon, Martin Burger, and Andreas Zeller. 2003. Automated debugging in Eclipse: (at the touch of not even a button). In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange - eclipse '03*. ACM Press, Anaheim, California, 1–5. https://doi.org/10.1145/965660.965661

[9] Michael D. Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Eric Kilmer, and Ronald Eytchison. 2024. A Broad Comparative Evaluation of Software Debloating Tools. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3927–3943. https://www.usenix.org/conference/usenixsecurity24/presentation/brown

[10] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. ACM, Montreal Canada, 1–5. https://doi.org/10.1145/1670412.1670413

[11] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. 2018. Reduce Before You Localize: Delta-Debugging and Spectrum-Based Fault Localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Memphis, TN, 184–191. https://doi.org/10.1109/ISSREW.2018.00005

[12] Holger Cleve and Andreas Zeller. 2000. Finding Failure Causes through Automated Testing. https://doi.org/10.48550/arXiv.cs/0012009 arXiv:cs/0012009.

[13] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 1017–1032. https://doi.org/10.1145/3453483.3454092

[14] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Tallinn Estonia, 729–739. https://doi.org/10.1145/3338906.3338972

[15] Raya Elsaleh and Guy Katz. 2023. DelBugV: Delta-Debugging Neural Network Verifiers. https://doi.org/10.48550/arXiv.2305.18558 arXiv:2305.18558 [cs].

[16] Alexander Elyasov, Wishnu Prasetya, Jurriaan Hage, and Andreas Nikas. 2014. Reduce first, debug later. In *Proceedings of the 9th International Workshop on Automation of Software Test*. ACM, Hyderabad India, 57–63. https://doi.org/10.1145/2593501.2593510

[17] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, Jun Sun, David Lo, and Yue Yu. 2023. Two Sides of the Same Coin: Exploiting the Impact of Identifiers in Neural Code Comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1933–1945. https://doi.org/10.1109/ICSE48619.2023.00164

[18] GCC Project. 2025. Reporting Bugs in GCC. https://gcc.gnu.org/bugs/. Accessed: 2025.

[19] Golnaz Gharachorlu and Nick Sumner. 2018. Avoiding the Familiar to Speed Up Test Case Reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Lisbon, 426–437. https://doi.org/10.1109/QRS.2018.00056

[20] Golnaz Gharachorlu and Nick Sumner. 2019. PARDIS: Priority Aware Test Case Reduction. In *Fundamental Approaches to Software Engineering*, Reiner Hähnle and Wil Van Der Aalst (Eds.). Vol. 11424. Springer International Publishing, Cham, 409–426. https://doi.org/10.1007/978-3-030-16722-6_24 Series Title: Lecture Notes in Computer Science.

[21] Golnaz Gharachorlu and Nick Sumner. 2021. Leveraging Models to Reduce Test Cases in Software Repositories. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, Madrid, Spain, 230–241. https://doi.org/10.1109/MSR52588.2021.00035

[22] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. 2020. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 237–248. https://doi.org/10.1145/3395363.3397349

[23] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause Reduction for Quick Testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, Cleveland, OH, USA, 243–252. https://doi.org/10.1109/ICST.2014.37

[24] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause reduction: delta debugging, even without bugs. *Software Testing, Verification and Reliability* 26, 1 (Jan. 2016), 40–68. https://doi.org/10.1002/stvr.1574

[25] Jason Gross, Théo Zimmermann, Rajashree Agrawal, and Adam Chlipala. 2025. Automatic Test-Case Reduction in Proof Assistants: A Case Study in Coq. https://doi.org/10.48550/arXiv.2202.13823 arXiv:2202.13823 [cs].

[26] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Long Beach CA USA, 263–272. https://doi.org/10.1145/1101908.1101948

[27] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. 2015. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, Bergamo Italy, 333–344. https://doi.org/10.1145/2786805.2786846

[28] Masatomo Hashimoto, Akira Mori, and Tomonori Izumida. 2018. Automated patch extraction via syntax- and semantics-aware Delta debugging on source code changes. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista FL USA, 598–609. https://doi.org/10.1145/3236024.3236047

[29] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto Canada, 380–394. https://doi.org/10.1145/3243734.3243838

[30] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana, IL, 861–871. https://doi.org/10.1109/ASE.2017.8115697

[31] Ralf Hildebrandt and Andreas Zeller. 2000. Simplifying Failure-Inducing Input. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*. ACM, 135–145.

https://doi.org/10.1145/347324.348938

[32] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. ACM, Seattle WA USA, 31–37. https://doi.org/10.1145/2994291.2994296

[33] Renáta Hodován and Ákos Kiss. 2016. Practical Improvements to the Minimizing Delta Debugging Algorithm:. In *Proceedings of the 11th International Joint Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, Lisbon, Portugal, 241–248. https://doi.org/10.5220/0005988602410248

[34] Renata Hodovan, Akos Kiss, and Tibor Gyimothy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Shanghai, 194–203. https://doi.org/10.1109/ICSME.2017.26

[35] Alexander Hoen, Dominik Kamp, and Ambros Gleixner. 2026. MIP-DD: Delta Debugging for Mixed-Integer Programming Solvers. *INFORMS Journal on Computing* 38, 2 (2026), 414–423. https://doi.org/10.1287/ijoc.2024.0844

[36] IEEE. 1991. Test Methods for Measuring Conformance to POSIX. ANSI/IEEE Standard 1003.3-1991; ISO/IEC Standard 13210:1994. Withdrawn 1997; superseded by IEEE Std 2003-1997 (ISO/IEC 13210:1999).

[37] Bo Jiang, Xiaoyan Wang, Huanqiang Xu, Hao Wang, and Chaoyang Zhang. 2018. Nondeterministic Event Sequence Reduction for Android Applications. In *2018 5th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, Dalian, China, 96–101. https://doi.org/10.1109/DSA.2018.00026

[38] Lingxiao Jiang and Zhendong Su. 2008. Profile-guided program simplification for effective testing and analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, Atlanta Georgia, 48–58. https://doi.org/10.1145/1453101.1453110

[39] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 321–322. https://doi.org/10.1109/SANER53432.2022.00048

[40] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Tallinn Estonia, 556–566. https://doi.org/10.1145/3338906.3338956

[41] Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 1003–1016. https://doi.org/10.1145/3453483.3454091

[42] Lukas Kirschner, Ezekiel Soremekun, and Andreas Zeller. 2020. Debugging inputs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 75–86. https://doi.org/10.1145/3377811.3380329

[43] Akos Kiss. 2020. Generalizing the Split Factor of the Minimizing Delta Debugging Algorithm. *IEEE Access* 8 (2020), 219837–219846. https://doi.org/10.1109/ACCESS.2020.3043027

[44] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical Delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, Lake Buena Vista FL USA, 16–22. https://doi.org/10.1145/3278186.3278189

[45] Gereon Kremer, Aina Niemetz, and Mathias Preiner. 2021. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Vol. 12760. Springer International Publishing, Cham, 231–242. https://doi.org/10.1007/978-3-030-81688-9_11 Series Title: Lecture Notes in Computer Science.

[46] Patrick Kreutzer, Tom Kunze, and Michael Philippsen. 2021. Test Case Reduction: A Framework, Benchmark, and Comparative Study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Luxembourg, 58–69. https://doi.org/10.1109/ICSME52107.2021.00012

[47] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[48] Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 1801–1825. https://doi.org/10.1145/3591294

[49] Jie Li, Changhai Nie, and Yu Lei. 2012. Improved Delta Debugging Based on Combinatorial Testing. In *2012 12th International Conference on Quality Software*. IEEE, Xi'an, Shaanxi, China, 102–105. https://doi.org/10.1109/QSIC.2012.28

[50] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-Based Debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, Buenos Aires, 393–403. https://doi.org/10.1109/ICSE.2017.43

[51] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov 2020), 1–25. https://doi.org/10.1145/3428264

[52] LLVM Project. 2025. How to Submit a LLVM Bug Report. https://llvm.org/docs/HowToSubmitABug.html. Accessed: 2025.

[53] Falin Luo, Mingwu Xue, Junqing Li, Keyao Li, Senyi Li, Gang Sun, and Hongfang Yu. 2023. Configuration Error Localization Based on Delta Debugging. In *2023 International Conference on Blockchain Technology and Information Security (ICBCTIS)*. IEEE, Xi'an, China, 27–32. https://doi.org/10.1109/ICBCTIS59921.2023.00011

[54] Konner Macias, Mihir Mathur, Bobby R. Bruce, Tianyi Zhang, and Miryung Kim. 2020. WebJShrink: a web service for debloating Java bytecode. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 1665–1669. https://doi.org/10.1145/3368089.3417934

[55] Mohamad Mansouri, Jun Xu, and Georgios Portokalidis. 2023. Eliminating Vulnerabilities by Disabling Unwanted Functionality in Binary Programs. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*. ACM, Melbourne VIC Australia, 259–273. https://doi.org/10.1145/3579856.3595796

[56] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. ACM, Shanghai China, 142–151. https://doi.org/10.1145/1134285.1134307

[57] Austin Mordahl, Dakota Soles, Miao Miao, Zenong Zhang, and Shiyi Wei. 2023. ECSTATIC: Automatic Configuration-Aware Testing and Debugging of Static Analysis Tools. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle WA USA, 1479–1482. https://doi.org/10.1145/3597926.3604918

[58] Yunbo Ni and Shaohua Li. 2025. Interleaving Large Language Models for Compiler Testing. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (Oct 2025), 815–841. https://doi.org/10.1145/3763079

[59] Tobias Paxian and Armin Biere. 2023. Uncovering and Classifying Bugs in MaxSAT Solvers through Fuzzing and Delta Debugging. In *Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023 (CEUR Workshop Proceedings, Vol. 3545)*. CEUR-WS.org, 59–71. https://ceur-ws.org/Vol-3545/paper5.pdf

[60] Yuanli Pei, Arpit Christi, Xiaoli Fern, Alex Groce, and Weng-Keen Wong. 2014. Taming a Fuzzer Using Delta Debugging Trails. In *2014 IEEE International Conference on Data Mining Workshop*. IEEE, Shenzhen, China, 840–843. https://doi.org/10.1109/ICDMW.2014.58

[61] Moritz Pflanzer, Alastair F. Donaldson, and Andrei Lascu. 2016. Automatic Test Case Reduction for OpenCL. In *Proceedings of the 4th International Workshop on OpenCL*. ACM, Vienna Austria, 1–12. https://doi.org/10.1145/2909437.2909439

[62] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2012. DARWIN: An approach to debugging evolving programs. *ACM Transactions on Software Engineering and Methodology* 21, 3 (June 2012), 1–29. https://doi.org/10.1145/2211616.2211622

[63] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens Greece, 441–452. https://doi.org/10.1145/3468264.3468539

[64] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-guided program reduction for understanding neural code intelligence models. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, San Diego CA USA, 70–79. https://doi.org/10.1145/3520312.3534869

[65] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards Causal Deep Learning for Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM, 1–11. https://doi.org/10.1145/3597503.3639170

[66] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, 335–346. https://doi.org/10.1145/2254064.2254104

[67] Jesse Ruderman. 2007. Lithium. https://github.com/MozillaSecurity/lithium.

[68] Joanna Sharrad and Olaf Chitil. 2021. Refining the Delta Debugging of Type Errors. In *33rd Symposium on Implementation and Application of Functional Languages*. ACM, Nijmegen Netherlands, 10–19. https://doi.org/10.1145/3544885.3544888

[69] Joanna Sharrad, Olaf Chitil, and Meng Wang. 2018. Delta Debugging Type Errors with a Blackbox Compiler. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. ACM, Lowell MA USA, 13–24. https://doi.org/10.1145/3310232.3310243

[70] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: a framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019). Association for Computing Machinery, 545–555. https://doi.org/10.1145/3338906.3338925

[71] Chaoxuan Shi, Tingwei Zhu, Tian Zhang, Jun Pang, and Minxue Pan. 2023. Structural-semantics Guided Program Simplification for Understanding Neural Code Intelligence Models. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. ACM, Hangzhou China, 1–11. https://doi.org/10.1145/3609437.3609438

[72] Rajvir Singh and Mamta Santosh. 2013. Test Case Minimization Techniques : A Review. *International Journal of Engineering Research* 2, 12 (2013).

[73] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. ACM, 1021–1031. https://doi.org/10.1145/3468264.3468589

[74] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. 2019. ReduKtor: How We Stopped Worrying About Bugs in Kotlin Compiler. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, CA, USA, 317–326. https://doi.org/10.1109/ASE.2019.00038

[75] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 203–213. https://doi.org/10.1145/2884781.2884879

[76] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[77] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 294–305. https://doi.org/10.1145/2931037.2931074

[78] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 361–371. https://doi.org/10.1145/3180155.3180236

[79] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A. Laredo, and Alessandro Morari. 2021. Probing model signal-awareness via prediction-preserving input minimization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens Greece, 945–955. https://doi.org/10.1145/3468264.3468545

[80] Sahil Suneja, Yufan Zhuang, Yunhui Zheng, Jim Laredo, Alessandro Morari, and Udayan Khurana. 2023. Code Vulnerability Detection via Signal-Aware Learning. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, Delft, Netherlands, 506–523. https://doi.org/10.1109/EuroSP57164.2023.00037

[81] Sahil Suneja, Yufan Zhuang, Yunhui Zheng, Jim Laredo, Alessandro Morari, and Udayan Khurana. 2023. Incorporating Signal Awareness in Source Code Modeling: An Application to Vulnerability Detection. *ACM Transactions on Software Engineering and Methodology* 32, 6 (Nov. 2023), 1–40. https://doi.org/10.1145/3597202

[82] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2022. XDebloat: Towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering* 48, 11 (Nov. 2022), 4501–4520. https://doi.org/10.1109/TSE.2021.3120213

[83] Yonggang Tao and Jingling Xue. 2025. Accelerating Delta Debugging through Probabilistic Monotonicity Assessment. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. ACM, Istanbul Turkiye, 12–22. https://doi.org/10.1145/3756681.3756940

[84] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 2023. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 2137–2141. https://doi.org/10.1145/3611643.3613101

[85] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 4873–4882. https://doi.org/10.24963/IJCAI.2023/542

[86] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2024. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 33, 1 (2024), 17:1–17:30. https://doi.org/10.1145/3617172

[87] Pablo Valle and Aitor Arrieta. 2022. Towards the Isolation of Failure-Inducing Inputs in Cyber-Physical Systems: is Delta Debugging Enough?. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Honolulu, HI, USA, 549–553. https://doi.org/10.1109/SANER53432.2022.00072

[88] Pablo Valle, Aitor Arrieta, and Maite Arratibel. 2023. Applying and Extending the Delta Debugging Algorithm for Elevator Dispatching Algorithms (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle WA USA, 1055–1067. https://doi.org/10.1145/3597926.3598117

[89] Dániel Vince. 2022. Iterating the minimizing Delta debugging algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. ACM, Singapore Singapore, 57–60. https://doi.org/10.1145/3548659.3561314

[90] Daniel Vince, Renata Hodovan, Daniella Barsony, and Akos Kiss. 2021. Extending Hierarchical Delta Debugging with Hoisting. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, Madrid, Spain, 60–69. https://doi.org/10.1109/AST52587.2021.00015

[91] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. 2022. The effect of hoisting on variants of Hierarchical Delta Debugging. *Journal of Software: Evolution and Process* 34, 11 (Nov. 2022), e2483. https://doi.org/10.1002/smr.2483

[92] Dániel Vince and Ákos Kiss. 2024. Evaluation of the fixed-point iteration of minimizing delta debugging. *Journal of Software: Evolution and Process* 36, 10 (Oct. 2024), e2702. https://doi.org/10.1002/smr.2702

[93] Dániel Vince and Ákos Kiss. 2024. GreeDDy: Accelerate Parallel DDMIN. In *Proceedings of the 15th ACM International Workshop on Automating Test Case Design, Selection and Evaluation*. ACM, Vienna Austria, 1–4. https://doi.org/10.1145/3678719.3685690

[94] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens Greece, 881–892. https://doi.org/10.1145/3468264.3468625

[95] Guancheng Wang, Yiqian Wu, Qihao Zhu, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2023. A Probabilistic Delta Debugging Approach for Abstract Syntax Trees. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Florence, Italy, 763–773. https://doi.org/10.1109/ISSRE59848.2023.00060

[96] Jie Wang. 2016. Constraint-based event trace reduction. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Seattle WA USA, 1106–1108. https://doi.org/10.1145/2950290.2983964

[97] Jie Wang, Wensheng Dou, Chushu Gao, Yu Gao, and Jun Wei. 2018. Context-Based Event Trace Reduction in Client-Side JavaScript Applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Vasteras, 127–138. https://doi.org/10.1109/ICST.2018.00022

[98] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 146–158. https://doi.org/10.1145/3575693.3575740

[99] Xiaoke Wang, Tao Hui, Lei Zhao, and Yueqiang Cheng. 2023. Input-Driven Dynamic Program Debloating for Code-Reuse Attack Mitigation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 934–946. https://doi.org/10.1145/3611643.3616274

[100] Yuanmin Xie, Zhenyang Xu, Yongqiang Tian, Min Zhou, Xintong Zhou, and Chengnian Sun. 2025. Kitten: A Simple Yet Effective Baseline for Evaluating LLM-Based Compiler Testing Techniques. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 25-28, 2025*, Mike Papadakis, Myra B. Cohen, and Paolo Tonella (Eds.). ACM, 21–25. https://doi.org/10.1145/3713081.3731731

[101] Qi Xin, Qirun Zhang, and Alessandro Orso. 2022. Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–13. https://doi.org/10.1145/3551349.3556970

[102] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, and Chengnian Sun. 2025. Boosting Program Reduction with the Missing Piece of Syntax-Guided Transformations. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (Oct. 2025), 86–112. https://doi.org/10.1145/3763053

[103] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, and Chengnian Sun. 2025. T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax. *ACM Transactions on Software Engineering and Methodology* 34, 2 (Feb. 2025), 1–31. https://doi.org/10.1145/3690631

[104] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 636–664. https://doi.org/10.1145/3586049

[105] Zhenyang Xu, Yiran Wang, Yongqiang Tian, Mengxiao Zhang, and Chengnian Sun. 2025. Latra: A Template-Based Language-Agnostic Transformation Framework for Effective Program Reduction. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16–20, 2025*. IEEE,

2274–2285. https://doi.org/10.1109/ASE63991.2025.00188

[106] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[107] Shin Yoo and Mark Harman. 2007. Pareto Efficient Multi-Objective Test Case Selection. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 140–150.

[108] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. 2012. Practical isolation of failure-inducing changes for debugging regression faults. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Essen Germany, 20–29. https://doi.org/10.1145/2351676.2351681

[109] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. 2012. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *Journal of Systems and Software* 85, 10 (Oct. 2012), 2305–2317. https://doi.org/10.1016/j.jss.2011.10.016

[110] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1687)*. Springer, 253–267. https://doi.org/10.1007/3-540-48166-4_16

[111] A. Zeller. 2001. Automated debugging: are we close? *Computer* 34, 11 (Nov. 2001), 26–31. https://doi.org/10.1109/2.963440

[112] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *SIGSOFT FSE 2002*. 1–10.

[113] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498

[114] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Vienna, Austria, 261–273. https://doi.org/10.1145/3650212.3652126

[115] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Xinru Cheng, and Chengnian Sun. 2025. Toward a Better Understanding of Probabilistic Delta Debugging. In *47th IEEE/ACM International Conference on Software Engineering*. IEEE, Ottawa, ON, Canada, 2024–2035. https://doi.org/10.1109/ICSE55347.2025.00117

[116] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 338–349. https://doi.org/10.1145/3611643.3616275

[117] Wentao Zhang, Jinghao Jia, Erkai Yu, Darko Marinov, and Tianyin Xu. 2025. DebCovDiff: Differential Testing of Coverage Measurement Tools on Real-World Projects. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1083–1094. https://doi.org/10.1109/ASE63991.2025.00094

[118] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 1073–1084. https://doi.org/10.1145/3540250.3549094

[119] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. 2018. Delta debugging microservice systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Montpellier France, 802–807. https://doi.org/10.1145/3238147.3240730

[120] Xintong Zhou, Zhenyang Xu, Mengxiao Zhang, Yongqiang Tian, and Chengnian Sun. 2025. WDD: Weighted Delta Debugging. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1592–1603. https://doi.org/10.1109/ICSE55347.2025.00071