Latra: A Template-Based Language-Agnostic Transformation Framework for Effective Program Reduction

Zhenyang Xu*, Yiran Wang*, Yongqiang Tian†, Mengxiao Zhang* and Chengnian Sun*

*University of Waterloo, Waterloo, Canada

Email: zhenyang.xu@uwaterloo.ca, y443wang@uwaterloo.ca, m492zhan@uwaterloo.ca, cnsun@uwaterloo.ca

†Monash University, Melbourne, Australia

Email: yongqiang.tian@monash.edu

Abstract—Essential for debugging compilers and interpreters, existing reduction tools face a fundamental trade-off. Language-specific reducers, such as C-Reduce and ddSMT, offer highly effective reductions but require substantial engineering effort for each target language. Conversely, language-agnostic reducers, like Vulcan, sacrifice effectiveness for broad applicability.

To bridge this gap, we present **Latra**, a novel template-based framework that balances both aspects, enabling general, effective, targeted program reduction. **Latra** combines language-agnostic reduction with user-defined, language-specific transformations. It facilitates user-defined transformations through a user-friendly domain-specific language based on simple matching and rewriting templates. This minimizes the need for deep formal grammar knowledge. **Latra** empowers users to tailor reductions to specific languages with reduced implementation overhead.

Our evaluation shows that **Latra** significantly outperforms Vulcan. On average, it reduces 33.77% more tokens in C and 9.17% more tokens in SMT-LIB, with 32.27% faster execution in SMT-LIB. Notably, **Latra** closely matches the effectiveness of language-specific reducers, i.e., C-Reduce and ddSMT (89 vs. 85, 103 vs. 109 tokens on average), while significantly reducing engineering effort (167 vs. 5,685, 62 vs. 118 lines of code). We strongly believe that **Latra** provides a practical and cost-efficient approach to program reduction, effectively balancing language-specific effectiveness with language-agnostic generality.

Index Terms—Program Reduction, Test Case Minimization

I. INTRODUCTION

Formal language tools, including compilers, interpreters, debuggers, and SMT solvers, are designed to process formal languages for purposes such as compilation, execution, analysis, and verification. These tools typically have extensive and complex codebases. The intricate nature of bugs within them presents significant challenges for developers during debugging. Programs that trigger such bugs can be extraordinarily large, sometimes spanning thousands of lines of code [1]–[9]. However, an empirical study [10] examining bug characteristics in GCC and LLVM compilers found that most bugs can be triggered by much smaller programs—often with fewer than 45 lines of code. In this context, program reduction [4], [11] can lower the debugging challenges. It automatically removes

irrelevant elements and condenses lengthy bug-triggering programs into more succinct versions that still trigger the same bugs. Many compiler and interpreter development communities, such as GCC, LLVM, JerryScript, and Mozilla, explicitly request bug reporters to utilize program reduction tools to reduce bug-triggering programs before reporting [12]–[15].

Program reduction tools can be broadly categorized into two types: language-agnostic reducers (AGRs) and language-specific reducers (SPRs), each with its own advantages and limitations.

Language-Agnostic Reducers (AGRs). AGRs, such as Delta Debugging (DD) [16], Hierarchical Delta Debugging (HDD) [17], Perses [4], and Vulcan [18], are able to reduce programs of any programming language. However, AGRs may not exhibit the most outstanding performance in terms of reduction size since they do not leverage language-specific knowledge. As a result, AGRs typically produce locally minimal results, where no single token can be removed on its own without losing the bug-triggering property. To approach global minimality, it requires effective yet complex language-specific transformations that AGRs do not perform due to their focus.

Language-Specific Reducers (SPRs). SPRs leverage the language-specific features and semantics to implement complex transformations. This involves the deletion or replacement of code fragments (mostly non-continuous tokens). These transformations can offer additional opportunities for reduction. Notable SPRs, such as C-Reduce [11] and ddSMT [19], incorporate C/C++ transformation rules using Clang Libtooling [20] and SMT-LIB formula mutators respectively into reduction algorithms to achieve significantly superior reductions. However, developing a new SPR from scratch can be challenging. It requires domain knowledge in program reduction, in-depth understanding of the language, and significant engineering effort for the development, testing, and ongoing maintenance of the tool. To implement transformations, users typically need to manually define node visitor patterns and perform Abstract Syntax Tree (AST) manipulations. For example, implementing the transformations in C-Reduce requires approximately 5,685 lines of code (LOC), while ddSMT re-

---

Zhenyang Xu and Yiran Wang contributed equally to this work.

quires 118 LOC. Additionally, users must be familiar with the internal infrastructure of ddSMT (1,200 LOC) to effectively develop transformations. Hence, we aim to decouple from toolchain-specific details to develop a tool that is more stable and maintainable.

Based on these observations, we intend to integrate the strengths of AGRs and SPRs, capitalizing on the generality of AGRs and the potent reduction capabilities of SPRs brought by language-specific transformations. To achieve this, we introduce Latra, a template-based, language-agnostic transformation framework for program reduction. It applies user-defined transformation rules to the reduced results of an AGR. To support these transformations, Latra provides a domain-specific language (DSL) that allows users to define program transformations in the format of match-rewrite template pairs. It decouples from concrete grammar rules with a code-like template syntax and releases users from the necessity of understanding the formal language grammar and internal structure of the tool. Latra employs a lightweight parse tree matching approach that leverages the match-rewrite template pairs to transform the bug-triggering programs minimized by an AGR. This may create additional reduction opportunities. The transformed program is then sent back to the AGR for another reduction trial. This iterative process continues until no further transformations and reductions can be achieved.

Latra offers several key advantages. First, Latra provides an intuitive and efficient approach to developing a ready-to-use SPR tailored to a specific language. It significantly reduces engineering effort compared to traditional SPRs from the users' perspectives. More specifically, Latra requires only the ANTLR grammar file for the language and transformation rules written in our DSL. This allows users to build an SPR framework without starting from scratch. Second, our approach decouples transformations from toolchain-specific dependencies. It can enhance the stability and maintainability. Since transformations are less sensitive to external library updates, the rule definitions remain consistent. Latra significantly reduces both engineering and future maintenance costs.

We evaluated Latra on 20 large C programs and 205 SMT-LIB benchmarks with AGRs such as Vulcan, and SPRs including C-Reduce and ddSMT. Latra reduced 33.77% more lexical tokens compared to Vulcan in C benchmarks, and 9.17% more tokens in SMT-LIB benchmarks, with 32.27% faster processing speed. It proves that Latra can further improve the performance of AGRs by incorporating language-specific transformations. When compared to the SPRs, Latra outperformed ddSMT in token size. Latra reduces to 103 tokens while ddSMT reduces to 109 tokens. This demonstrates Latra's ability to achieve substantial reduction, reaching similar or more profound reduction results than SPRs. Moreover, Latra reaches a close reduction result as C-Reduce. It is able to match or exceed the reduction performance with a lower engineering effort. Latra streamlines rule definition process and requires up to 34× fewer lines of code compared to C-Reduce.

The main contributions of this work are as follows:

- We propose Latra, a language-agnostic reduction framework that enables language-specific transformations to be implemented on an AGR using match-rewrite templates to enhance AGR performance.
- We implemented transformations in C and SMT-LIB to demonstrate the capabilities of Latra. To support replication, reproducibility, and to benefit various language tool communities, we have provided open access to the artifact at https://github.com/uw-pluverse/latra-artifact.
- We evaluated Latra across multiple programming languages with program reducers including both AGRs and SPRs. The results demonstrate Latra's effectiveness, efficiency, and low engineering cost to construct transformations.

II. MOTIVATING EXAMPLE

```
1  (declare-fun
2    bvlambda_3_ ()
3    (_ BitVec 1))
4  (declare-fun a88 ()
5    (Array (_ BitVec 1
6      (_ BitVec 5)))
7
8  (assert
9    (let
10     (($e30 (_ bv0 5)))
11     (let (($e341
12        (store a88
13          (not false)
14          $e30
15        )
16     ))
17     (let (($e342
18        (select $e341
19          (not
20            bvlambda_3_)
21        )
22     )) (extract $e342))
23     )
24   )
25  )
```

(a) Result by Vulcan.

```
1  (declare-fun
2    bvlambda_3_ ()
3    (_ BitVec 1))
4  (declare-fun a88 ()
5    (Array (_ BitVec 1
6      (_ BitVec 5)))
7
8  (assert
9    (extract
10     (select
11       (store a88
12         (not false)
13         (_ bv0 5)
14       )
15       (not bvlambda_3_)
16     )
17   )
18  )
```

(b) Transformed program from (a) after applying let substitution transformation rule in (c) by Latra.

```
rule LetSubstitution {
    from (let ((:[sym] :[tem+])) :[body+])
    such_as (let ((a (+ x 5)) (* a 2))
    to :[body]
    global_replace :[sym] inside :[body] with :[tem]
}
```

(c) The transformation rule for substituting `let` keywords using the matching (`from` clause, `such_as` clause) and rewriting (`to` clause, `global_replace` clause) templates.

Fig. 1: An example in SMT-LIB that performs substitution of let keywords. (a) is the minimal bug-triggering result produced by an AGR, Vulcan. (b) is obtained after applying the transformation rule described in (c). (c) is the user-defined transformation rule in Latra for removing `let` keywords and replacing the bounded variables by the underlying values.

To demonstrate how Latra works, we use a real-world compiler bug from the SMT-LIB benchmark suite

(`btor2-bug-12208-547`) [21] to illustrate. This bug triggers a crash of `boolector-1.5.125`, a SMT-LIB solver, due to an assertion failure in `btorexp.c`.

Figure 1a presents the minimized bug-triggering program generated by Vulcan that guarantees 1-tree-minimality. The program contains three nested `let` expressions. In SMT-LIB, the `let` expression creates local bindings for terms. This allows bound variables to be referenced within the body of the expression for better readability. For example, (`let ((a (+ x 5))) (* a 2))` binds the variable `a` to `(+ x 5)`. The variable is then used in the body of the expression (i.e., `(* a 2)`). However, in Figure 1a, the three nested `let` expressions overcomplicate the logic because their bound variables are only referenced once in the body. Therefore, substituting bound values directly into the body of the expression can make the program more concise. With Latra, this substituting transformation can be implemented through a user-defined transformation rule as shown in Figure 1c, and eventually, Latra can further reduce the program size from 83 tokens (after the minimization by the AGR Vulcan) to 60 tokens.

The remaining section introduces two core concepts of Latra, i.e., matching template and rewriting template, and discusses the benefits of Latra in program reduction.

A. Matching Template

The key component of a matching template is a pattern (i.e., the `from` clause in our DSL detailed in § III-B) that specifies the code segments in the input program P that should be matched and transformed. We adopt the syntax from Comby [22] and PolyglotPiranha [23]. Each pattern is a sequence that contains concrete tokens and/or holes. A concrete token (e.g., `let`) matches only identical tokens in the input program, whereas a hole (e.g., `:[sym]`) matches any token and tags the matched token with a user-defined label (i.e., `sym`). Latra also supports greedy holes, e.g., `:[tem+]`, `:[body+]`. This type of holes can match multiple tokens (e.g., `(extract $e342)`) and tag all the matched tokens with a specified label (i.e., `body`). All labels in the pattern can subsequently be referenced in the rewriting template to denote how matched tokens should be modified. In the matching template shown in Figure 1c, the pattern is defined as (`let ((:[sym] :[tem+])) :[body+])`. It retains the concrete `let` keyword and pairs of parentheses, which means that they need to be matched exactly. The symbol names, terms and the body are generalized using holes so this pattern can match all `let` expressions regardless of the variables and content. In one transformation cycle, a `let` expression, highlighted in blue, is found to match the matching pattern. The label `sym` is bound to the variable `$e342`. Similarly, the label `tem` is bound to (`select $e341 (not bvlambda_3_))`, and `body` is bound to (`extract $e342`).

A matching template can optionally include an example (i.e., the `such_as` clause). It is a list of concrete tokens that can be matched by the pattern, e.g., (`let ((a (+ x 5))) (* a 2))`. Including an example in the matching template can improve both the efficiency and accuracy of matching. The details are discussed in § III-C.

B. Rewriting Template

A rewriting template consists of a pattern (i.e., the `to` clause) and optional operations (e.g., the `global_replace` clause). The pattern specifies how the matched tokens in the input program should be rewritten. It contains a list of concrete tokens and/or holes, similar to patterns in matching templates. However, any hole label used in the rewriting pattern must first be defined in the pattern of the corresponding matching template. For each rewriting, the pattern is concretized by substituting each hole with the tokens tagged with the corresponding label from the matching process. The matched code snippets in the program are then replaced with this concretized pattern. A rewriting template pattern can be empty. In this case, Latra deletes all matched tokens.

In this `LetSubstitution` transformation, the rewriting pattern is `:[body]` (i.e., `(* a 2)`), as the local binding will be unnecessary and can be removed after substituting the bound value into the body. To perform the substitution, we use `global_replace` clause to replace `:[sym]` (i.e., `$e342` in this matched case) that appears within `:[body]` (i.e., `(extract $e342)`) by `:[tem]` (i.e., `(select $e341 (not bvlambda_3_))`). The `inside` clause helps limit the scope of the replacement. As a result, the matched `let` expression highlighted in blue is transformed into (`extract (select $e341 (not bvlambda_3_)))`. This becomes the new body of the surrounding `let` expression. A new match for the `let` substitution transformation would then be identified. In some cases, this transformation may temporarily increase program size. However, Latra does not reject such edits since they may introduce further reduction opportunities. The process continues iteratively until no further matches are found. Ultimately, the program in Figure 1a is transformed into the one shown in Figure 1b, significantly reducing the number of tokens.

C. Benefits of Using Latra

By automating the matching and rewriting processes with user-defined transformation rules in our DSL, Latra simplifies the SPR development and eliminates the need for deep expertise in the language grammar, AST traversal, and toolchain-specific infrastructures.

Ease of Use. ddSMT requires users to learn its SMT-LIB-specific infrastructure and demands familiarity with SMT-LIB formula construction to implement transformations. Similarly, C-Reduce requires working directly with the internal structure of LLVM. It requires expertise in compiler toolchains, type systems, and AST manipulation. Latra abstracts these complexities and simplifies the development of language-specific transformation rules by allowing users to define matching and rewriting templates through its DSL. The transformation code is automatically generated based on the rule definitions. It would minimize coding effort and eliminate the need for deep

expertise in AST patterns, traversal, and manipulation. Furthermore, by leveraging parse trees, Latra avoids the overhead of visitor patterns and manual AST traversal. This ensures greater adaptability across languages while maintaining precise transformation control. Although a basic understanding of the target language remains necessary, Latra significantly reduces trial-and-error costs to implement transformations. The transformation process is thus more intuitive and efficient.

Maintainability. Latra's decoupling from toolchain-specific dependencies enhances stability and maintainability. In contrast, C-Reduce relies on Clang LibTooling [20] and must continuously adapt to new LLVM releases. Developers typically update C-Reduce between two to ten times per year, with each update involving an average of 80 lines of code changes, sometimes exceeding 250 lines for major LLVM versions (e.g. LLVM 7.0, LLVM 8.0). Unlike C-Reduce, which is tightly integrated with the evolving architecture of LLVM, Latra interacts with public APIs. This reduces the impact of external library updates and minimizes the need for frequent modifications when dependencies change. Additionally, our transformation rules are decoupled from concrete grammar structures. They do not directly involve specific non-terminal symbols. As a result, changes in the concrete grammar, such as renaming `declaration` to `decl`, do not require updates to Latra's transformation rules. This improves maintainability and reduces development effort.

III. APPROACH

This section presents the workflow of Latra with a detailed explanation of the framework.

A. Workflow

Figure 2 shows the workflow of Latra. Step ② in the figure is the typical workflow of using an AGR to reduce a bug-triggering program. To further reduce the result produced by an AGR, Latra allows users to define language-specific transformation rules for any programming language using a specifically designed DSL (details of the DSL are in § III-B). Then, Latra synthesizes the corresponding code for matching and rewriting for each defined transformation rule (step ①), and performs the transformations to directly reduce the result produced by the AGR and/or create reduction opportunities for the AGR to further reduce the program (step ③, ④, and ⑤). Finally, the transformed program is sent back to the AGR and the process (step ⑥) is repeated until no progress can be made.

Reducing Using an AGR. In Latra, a bug-triggering program is first reduced by an AGR (Vulcan is adopted in the prototype). Specifically, a bug-triggering program P is first passed into the AGR along with a property checker ψ that checks whether a program still triggers the specific bug. Since the recently proposed AGRs such as Perses and Vulcan perform reduction on the parse tree of the program, a reduced parse tree that represents the reduced program $\overline{P}$ is obtained after the reduction (step ②). Typically, at the start of the reduction process, the program contains a relatively large amount of

bug-irrelevant content, much of which can be eliminated by the state-of-the-art AGR. Therefore, reducing with an AGR first can efficiently reduce the size of the input program to a relative small scale, making subsequent language-specific transformations significantly more computationally efficient.

Performing User-Defined Transformations. Once a reduced parse tree is produced by the AGR, Latra starts a matching and rewriting process to perform the user-defined transformations. Specifically, Latra first traverses the parse tree in a depth-first manner to find a match based on the matching template defined in the transformation rule (step ③). Then, the matched content in the parse tree is written according to the rewriting template (step ④). For each rewriting, the resulted program P' is checked with the property checker ψ and if the P' exhibits the property, P' will be used to replace $\overline{P}$ for subsequent reduction (step ⑤). This process is repeated until there is no rewriting for any user-defined transformations that can be performed without losing the property.

Iterating Until Fixpoint. The transformed program is then sent back to the AGR for further reduction (step ⑥). This iterative process ensures that any potential reduction opportunities generated by user-defined transformations, can be seized by the AGR. If the AGR reduces the program further, any new rewriting opportunities created by the AGR can be leveraged by Latra. The iteration continues until neither the AGR nor the transformations can make further progress.

B. Language Design

The grammar of the DSL used for constructing transformation rules in Latra is shown in Figure 3. As shown in the figure, a transformation rule (i.e., ⟨rule⟩) consists of a `rule` keyword, a ⟨label⟩ as the name of the transformation, and a pair of ⟨matching_template⟩ and ⟨rewriting_template⟩ wrapped in curly braces.

1) Matching Template: The matching template defines the code snippets that a transformation identifies and subsequently transforms. It consists of the following components.

Matching Pattern. A matching pattern (i.e., the `from` clause) is required for every transformation rule. It defines the code segments the transformation would identify. As illustrated in Figure 3, each matching pattern consists of a list of ⟨terms⟩, where each term can be a concrete token ⟨token⟩, a hole :[⟨label⟩], or a greedy hole :[⟨label⟩+]. Concrete tokens can only match identical tokens in the program, whereas holes can match any token. A non-greedy hole matches a single token, while a greedy hole matches multiple tokens. Tokens matched by a hole are bound to the hole's label. It can subsequently be referenced in the rewriting template to modify the corresponding matched tokens.

Matching Example. A matching example (i.e., the `such_as` clause) is an optional component. It specifies a valid code instance that can be matched by the pattern. This facilitates the process and accuracy in determining the types of subtrees where a match can be found. In our prototype, 8 of our 27 C transformations and 1 of the 12 SMT-LIB transformation use examples. Removing the matching example
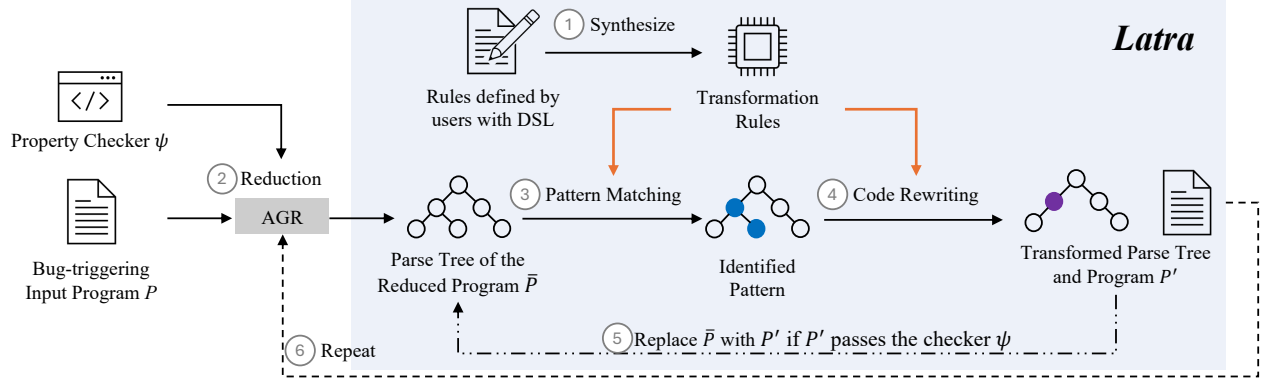
Fig. 2: The workflow of Latra.

$\langle rule \rangle$ ::= **'rule'** $\langle label \rangle$ **'{'**
$\quad\quad\quad\quad$ $\langle matching\_template \rangle$
$\quad\quad\quad\quad$ $\langle rewriting\_template \rangle$
$\quad\quad\quad\quad$ **'}'**
$\langle matching\_template \rangle$ ::= **'from'** $\langle pattern \rangle$
$\quad\quad\quad\quad$ (**'such_as'** $\langle token \rangle$+)?
$\langle rewriting\_template \rangle$ ::= **'to'** $\langle pattern \rangle$?
$\quad\quad\quad\quad$ (**'insert'** $\langle pattern \rangle$ $\langle location \rangle$)?
$\quad\quad\quad\quad$ (**'global_replace'** $\langle pattern \rangle$
$\quad\quad\quad\quad\quad$ (**'such_as'** $\langle token \rangle$+)?
$\quad\quad\quad\quad\quad$ (**'inside'** $\langle label \rangle$)
$\quad\quad\quad\quad\quad$ **'with'** $\langle pattern \rangle$)∗
$\langle pattern \rangle$ ::= $\langle terms \rangle$+
$\langle term \rangle$ ::= **':['** $\langle label \rangle$ **'+'** ? **']'**
$\quad\quad\quad$ | $\langle token \rangle$
$\langle location \rangle$ ::= **'top'** | **'bottom'** |
$\quad\quad\quad\quad$ **'before'** | **'after'**
$\langle label \rangle$ ::= $[a - zA - Z\_][a - zA - Z0 - 9\_]*$
$\langle token \rangle$ ::= a token in the target language

Fig. 3: The grammar of the Latra DSL for specifying program transformations.

for these transformations impairs their matching precisions, which in turn compromises reduction performance by generating uninteresting program variants.

2) Rewriting Template: The rewriting template specifies the edits applied to the program after a match is found. It consists of a rewriting pattern and optional operations.

Rewriting Pattern. The rewriting pattern, defined using the `to` clause, specifies how the matched instance should be transformed. Its syntax follows that of the matching pattern, with the restriction that only labels defined within the matching pattern can be referenced. If the rewriting pattern is left empty, it indicates that the matched instance is entirely removed.

Operation. A rewriting template can have operations to

perform additional code manipulation.

- The `insert` clause relocates code snippets to designated positions within the program, such as the top, bottom, or positions before or after the match-rewrite point. This applies for transformations that require code reordering or structural modifications.

- The `global_replace` clause performs substitution. The target code snippets to be replaced is a $\langle pattern \rangle$, consisting of concrete tokens and labels (either predefined or newly introduced). The `inside` clause is optional and specifies the scope of transformation within the program. By default, substitutions are applied to all occurrences of the matched pattern in the program.

C. Parse Tree-Based Two-Level Matching

Algorithm 1: Parse Tree-Based Two-Level Matching Algorithm (`ComputeMatch` is defined in Algorithm 2)

Input: `root`, the root node of the reduced parse tree from the input program P.
Input: `pattern`, the pattern for code matching.
Input: `example`, an optional matching example specified in the `such_as`, defaults to an empty string.
Output: `matches`, a set of mappings that map labels to tokens

1   compatible_rules ← `GetCompatibleRules`(example)
2   stack ← a stack containing root
3   matches ← ∅
4   **while** |stack| ≠ 0 **do**
5   node ← stack.pop()
6   **if** |compatible_rules| = 0 ∨ node.rule ∈ compatible_rules **then**
7   match ← `ComputeMatch`(node, pattern)
8   matches = matches ∪ match
9   **foreach** child *in* node.children **do** stack.push(child)

10   **return** matches

To identify matches in the input program P, Latra traverses its parse tree and checks whether the leaf nodes of each subtree match the specified pattern. However, a significant number of subtrees are typically irrelevant. To enhance efficiency, Latra

employs a parse tree-based two-level matching algorithm. Algorithm 1 provides a detailed description of this approach.

Searching for Possible Subtrees. Latra first identifies subtrees where a match is potentially feasible. This pre-filtering is enabled if a `such_as` clause is provided in the matching template. This allows Latra to determine the production rules that can parse the example (line 1). Subsequently, only subtrees corresponding to these compatible rules are considered for matching. For instance, the matching example `(let ((a (+ x 5))) (* a 2))` in Figure 1c can be parsed by ANTLR grammar rule `<term>`. Latra then only evaluates subtrees of node with the rule `<term>`. The lines 1 to 9 describes the first level of matching. Once a compatible rule list, which can be empty, is obtained, Latra traverses the parse tree in a depth-first manner. For each node, Latra first verifies whether the grammar rule of the node is in the compatible rule list (line 6). If the list is not empty and it is not in the list, the node is skipped. Otherwise, Latra invokes `ComputeMatch` to conduct the second-level matching with the subtree (line 7). If a match is found in the second-level matching, it is added to the set for potential transformation.

Algorithm 2: `ComputeMatch()`

Input: node, a node of the reduced parse tree from the input program P.
Input: pattern, the pattern for code matching.
Output: match, a map that map labels to tokens if a match is found

1 leaves ← node.leaves
2 token_index ← 0
3 holes_to_match ← []
4 match ← ∅
5 **foreach** element *in* pattern **do**
6   **if** element *is hole* **then**
7   holes_to_match.append(element)
8   **continue**
9   **if** element ∉ leaves[token_index :] **then return** ∅
10   next_index ← leaves[token_index + 1 :].FindNextBalancedToken(element) + 1
11   tokens_to_match ← leaves[token_index + 1 : next_index − 1]
12   **if** |holes_to_match| = 0 **then**
13   token_index ← next_index
14   **continue**
15   **if** |holes_to_match| > |tokens_to_match| **then return** ∅
16   hole_token_pairs ← MatchHoles(holes_to_match, tokens_to_match)
17   match ← match ∪ hole_token_pairs
18   token_index ← next_index
19   holes_to_match ← []
20 **if** |holes_to_match| ≠ 0 **then**
21   tokens_to_match ← leaves[token_index :]
22   **if** |holes_to_match| > |tokens_to_match| **then return** ∅
23   hole_token_pairs ← MatchHoles(holes_to_match, tokens_to_match)
24   match ← match ∪ hole_token_pairs
25 **if** *any leaf node is not visited* **then return** ∅
26 **return** match

Matching in Possible Subtrees. The function shown in

Algorithm 2 describes the second-level matching process, which attempts to find a match between leaf nodes of a possible subtree and the matching pattern. At the beginning, the function initializes `leaves` with the leaf nodes of the given subtree, sets `token_index` (which is maintained to point to the first unvisited token) to 0, and creates an empty list for temporarily storing holes to be matched in the pattern and a map for storing the match (line 4). Next, the function traverses the elements in the matching pattern (line 5). For each element, if it is a hole, it will be temporarily appended to the list, `holes_to_match` (line 8). Otherwise, if the element is a concrete token, the function first checks its presence within the unvisited leaf nodes of the subtree and returns an empty map, indicating no match can be found, if there is no such a concrete token in the remaining leaf nodes (line 9). Next, if the concrete token exists in the remaining leaf nodes, the index of the concrete token is obtained, and `next_index` is pointed to the next token of the concrete token. It should be noted that for concrete tokens that appear in pairs (e.g., `")"`, `"]"`, and `"}"`), the function `FindNextBalancedToken` obtains the index of the next balanced one (line 10). Then, all the leaf nodes before the concrete token that has not been matched is stored to `tokens_to_match` (line 11). If `tokens_to_match` is empty, `token_index` is updated with `next_index`, and the function terminates the current iteration and moves to the next element in `pattern` (lines 12 to 14). Otherwise, the function attempts to match `tokens_to_match` with `holes_to_match` (lines 15 to 19). If `holes_to_match` is empty or the number of holes it contains is larger than the number of tokens in `tokens_to_match`, then an empty map will be returned as it is impossible to match (line 15). Otherwise, `MatchHoles` is invoked to match holes with tokens and update `match` (line 16). If the matching is successful, `token_index` will be updated with `next_index`, and `holes_to_match` will be cleared (lines 17 to 19). Otherwise, an empty map will be returned. After the `foreach` loop, if `holes_to_match` is not empty, the function needs to perform a matching for the remaining holes (lines 20 to 24). Eventually, after all the holes are matched, if there is still any leaf node that has not been visited, the matching is failed and an empty map is returned (line 26).

D. Rewriting Process

After matches are identified, Latra applies the corresponding rewriting template to transform the program. For each matched code instance, Latra follows the rewriting pattern. The pattern can either be empty, indicating that the matched code should be deleted, or contain concrete tokens and labels. The DSL also supports operation clauses for more precise token substitution and relocation. Labels previously defined in the matching pattern, or newly defined labels in the substitution pattern, are concretized by their corresponding matched tokens before the program edit is applied to the program. The program is then re-constructed by deleting, replacing the matched code snippets, or adding and inserting new snippets

at the targeted node locations as matched or indicated. Then, Latra re-parses the program. The edit will be applied to the program if it can be parsed successfully. The transformed program is then passed to the property checker for property testing. If it fails the property testing, the transformation will not be applied to the program and Latra will proceed to the next potential match until a transformation can be applied. Finally, the transformed program input is sent back to the AGR for additional reduction checks.

Note that Latra does not automatically enforce the syntactic or semantic validity of the resulting transformed program. The primary goal is to provide a flexible mechanism to specify diverse program transformations for further minimizing programs. Depending on the use cases of reduction, users may have various requirements regarding the validity of the resulting programs. For example, a minimized, crash-inducing program might not require semantic validity, whereas one reduced to isolate an optimization bug typically would. As a general-purpose reduction framework, Latra operates without predefined knowledge of the specific properties under consideration and it is difficult for Latra to universally enforce syntactic or semantic correctness.

E. Limitations

Latra adopts a lightweight approach by performing pattern matching and rewriting on the parse tree of the program rather than the AST. Although this approach enables Latra to easily support different programming languages, some inherent limitations also exist. Nodes in AST contain semantic information of the program such as types, but a parse tree is simply a syntactic representation. Hence, it is challenging, or even infeasible, for Latra to implement transformations requiring semantic information or complex domain knowledge of a certain language. Specifically, for the C-specific program transformations in C-Reduce, Latra is not able to implement 6 of them, and among 53 transformations in ddSMT, 18 of them cannot be faithfully expressed by Latra. The main reason is that these transformations require semantic information. For example, `RemovePointer`, a transformation in C-Reduce, reduces the level of a pointer if this pointer is not referenced or dereferenced. Performing this transformation requires program analysis and type information, which is not supported by Latra. Another example is the `ArithmeticSimplifyConstant` transformation in ddSMT. This transformation replaces numeric constants with smaller values by repeatedly dividing them by 2, which requires runtime evaluation, which Latra does not support.

However, although there are several transformations that Latra cannot faithfully express, some of them can be approximated in the Latra framework. For example, for the `ArithmeticSimplifyConstant` transformation, although Latra cannot perform the exact same transformation, it can support a similar transformation, i.e., directly replacing constants with other constants (e.g., 0 or 1). Moreover, it is *important* to note that the goal of Latra is to provide a program transformation framework that enables rapid and cost-efficient development of language-specific transformations for effective program reduction, rather than to serve as a comprehensive solution for complex, heavyweight program transformations like those in C-Reduce.

IV. EVALUATION

We evaluate Latra with the following research questions.
RQ1: How effective is Latra versus other program reducers?
RQ2: How efficient is Latra versus other program reducers?
RQ3: What is the engineering effort required for users to define transformation rules?

A. Experimental Setup

Latra incorporates Vulcan as the AGR due to its outstanding performance in language-agnostic reduction. Built on top of Perses, Vulcan utilizes language-agnostic transformations to push the reduction results. However, if the search space (i.e., program size) is large, Vulcan may experience longer runtimes. Latra is not tied to Vulcan and can be integrated with other AGRs. To maximize efficiency, Latra first applies transformations on the reduced parse tree from Perses. Then it passes the minimized program to Vulcan for further reduction. In this way, we can potentially reduce runtime.

Our evaluation focuses on C and SMT-LIB to demonstrate Latra's applicability across different languages. These languages are chosen because their respective SPRs, C-Reduce and ddSMT, outperform Vulcan. We used the C benchmark suite from Vulcan [18] and other related works [4], [24], [25], with 20 large real-world bug-triggering programs in GCC and Clang compilers, and 205 SMT-LIB programs from ddSMT [19]. The experiments are conducted on an Ubuntu 22.04 server equipped with Intel Xeon Gold 6348 CPU and 512 GB memory. For a fair comparison, each reducer operated in a single-thread environment throughout the experiments.

To evaluate Latra, we implemented 27 C and 12 SMT-LIB transformation rules from existing SPRs. Due to the distinctions in framework design between Latra and these SPRs, and a lack of semantic information (see § III-E), we only considered transformations that are feasible to implement. The list of the selected transformations is shown in Table I. In terms of the priority of these transformations (i.e., which transformation to perform first), we follow the order in C-Reduce and ddSMT, presuming their order is optimized. During reduction, when multiple transformation are applicable, the first one is performed.

B. RQ1: Effectiveness of Latra

We compared Latra to both AGRs such as Vulcan, and SPRs including C-Reduce and ddSMT for an analysis of Latra's reduction performance among the program reducers. We use the following metrics to assess the effectiveness of the tools:
1) **R(#):** the final number of lexical tokens in the reduced programs; the smaller the R(#) is, the more effective the reducer is.
2) **C(%) w.r.t. the reducer:** the percentage of lexical tokens reduced by Latra compared to each baseline reducer, calculated as $C(\%) = \frac{R_{Latra} - R_{baseline}}{R_{baseline}} \times 100\%$, where R_{Latra}

TABLE I: The transformation rules implemented in Latra for the evaluation, and the respective engineering cost of C-Reduce, ddSMT, and Latra to construct the rules in terms of lines of code (LOC). Note that the asterisk (*) indicates that the implementations are written within the same function or class.

Lang.	Transformation Rule	LOC in SPRs	LOC in Latra
C	aggregate-to-scalar	256	7
	simplify-operation	130	5
	replace-call-by-zero	91	5
	combine-variables	177	6
	copy-propagation	343	6
	empty-struct-to-int	429	6
	lift-assignment	174	5
	local-to-global	205	5
	reduce-array-dim	283	6
	remove-addr-taken	123	6
	remove-array	213	6
	replace-array-access	88	5
	replace-definition-by-declaration	253	5
	replace-undefined-function	120	6
	simplify-call-expr	124	7
	simplify-comma-expr	127	5
	simplify-if	120	10
	simplify-struct-union-decl	170	12
	simplify-struct	245	6
	unify-function-declaration	110	6
	union-to-struct	318	6
	inline-function	501	6
	remove-typedef	359	6
	void-declaration void-definition	*189	5 5
	replace-call-by-return-value	457	8
	move-definition-to-declaration	80	6
SMT-LIB	remove-annotation	7	5
	convert-bv	8	5
	simplify-check-sat	7	5
	simplify-double-neg simplify-double-not	*11	5 5
	remove-false-check	8	5
	remove-exists remove-forall	*18	5 5
	substitute-let	22	6
	merge-and merge-plus	*10	5 5
	replace-variable	27	6

and $R_{baseline}$ denote the size of the results produced by Latra and a baseline, respectively.

Performance on C Benchmarks. The resulting token size, query amount and time spent for Vulcan, C-Reduce and Latra to run each test case are summarized in Table II. Latra is capable of reducing more tokens than Vulcan across all the benchmarks we have. On average, Vulcan reduces the number of lexical tokens in the C benchmarks from 94,487 to 151. Latra, leveraging the 27 C-specific transformations listed in Table I and the iterative nature of the reduction framework with Vulcan, can further reduce the token amount to 89. There

is an average of 33.77% more tokens being reduced by Latra. We then conduct a sign test for the reduction results. The p-value of 4.77×10^{-7} that is smaller than 0.05 indicates that Latra significantly outperforms Vulcan in these C benchmarks statistically. By introducing language-specific transformations, Latra further pushes the reduction results of an AGR.

Among the 20 benchmarks, Latra outperforms C-Reduce in 11 benchmarks as highlighted. It demonstrates the capability to achieve comparable or superior reduction results. Although the average number of tokens reduced (89 tokens) is slightly larger than that of C-Reduce (85 tokens), the sign test gives a p-value of 0.52. It shows that the difference is not statistically significant. Latra and C-Reduce perform similarly overall in terms of reduction effectiveness. By implementing language-specific transformations on an AGR, Latra can achieve comparable results as the SPR.

Performance on SMT-LIB Benchmarks. The reduction results of Vulcan, ddSMT, and Latra on SMT-LIB programs are shown in Figure 4. Specifically, Figure 4a visualizes the distribution of the reduction size of the three reducers for comparison using boxplots. Overall, Latra possesses greater reduction abilities over both the AGR, Vulcan, and the SPR, ddsmt. On average, Latra reduces the 205 SMT-LIB benchmarks from 28,179 tokens to 103 tokens. In contrast, Vulcan only reduces the average size to 121 tokens. The results of Latra are 9.17% smaller than that of Vulcan as shown in Figure 4b. Meanwhile, ddSMT reduces the programs to 109 tokens, which is slightly larger than Latra. We further conduct hypothesis testing to investigate if the reduction sizes of Latra is smaller than Vulcan and ddSMT in statistics. The p-values are 4.65×10^{-25} and 1.42×10^{-5} for Vulcan and ddSMT, respectively, and both are smaller than 0.05. The results indicate that Latra significantly outperforms Vulcan and ddSMT in statistics in terms of reduction size. By leveraging the 12 SMT-LIB transformation rules implemented, it demonstrates that Latra provides additional reduction opportunities across all programs to further reduce the program size, and its overall performance is in the same level as ddSMT.

> **RQ1**: Latra improves the AGR performance by reducing 33.77% and 9.17% more tokens than Vulcan on average in C and SMT-LIB benchmarks. Although C-Reduce reduces slightly more tokens on average, this advantage is not statistically significant. For SMT-LIB, Latra outperforms ddSMT, which demonstrates its potential to bridge the gap between AGRs and SPRs.

C. RQ2: Efficiency of Latra

To study the efficiency of Latra in program reduction compared to other program reducers, we use the following metrics to measure the efficiency of the reduction technique:
1) **Q(#):** the amount of property tests (queries) executed and;
2) **T(s):** the duration of the reduction process in seconds.

Comparison with AGRs. Latra does not outperform Vulcan on C benchmarks but exhibits a more efficient reduction

TABLE II: The reduction results of Vulcan, C-Reduce, and Latra on the C benchmarks. **O(#)** denotes the original number of tokens in the benchmark. **R(#)** denotes the remaining number of lexical tokens after the reduction by the tool. **Q(#)** denotes the number of property tests or queries that have been executed. **T(s)** denotes the amount of time in seconds that is required to complete the reduction. Cells in green indicate the best reduction results among the tools.

Bug	O(#)	Vulcan			C-Reduce			Latra			C(%) w.r.t.	
		R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	Vulcan	C-Reduce
clang-22382	21,068	108	10,905	830	70	13,186	938	107	27,916	1,509	-0.93%	52.86%
clang-22704	184,444	62	5,353	1,245	42	11,189	1,623	54	14,194	2,240	-12.90%	28.57%
clang-23309	38,647	303	56,576	6,434	118	32,167	3,290	100	44,506	6,564	-67.00%	-15.25%
clang-23353	30,196	91	8,477	793	74	12,299	966	70	19,367	1,948	-23.08%	-5.41%
clang-25900	78,960	104	10,308	1,362	90	16,517	1,459	83	24,642	2,632	-20.19%	-7.78%
clang-26760	209,577	56	7,112	1,934	43	12,992	2,358	42	12,808	3,169	-25.00%	-2.33%
clang-27137	174,538	88	14,258	9,463	50	25,049	8,604	54	16,927	13,038	-38.64%	8.00%
clang-27747	173,840	79	7,382	1,437	68	15,138	1,703	63	20,768	2,993	-20.25%	-7.35%
clang-31259	48,799	282	51,245	11,429	168	26,997	4,383	130	92,445	20,394	-53.90%	-22.62%
gcc-59903	57,581	198	22,432	4,235	105	48,652	5,438	177	71,309	8,310	-10.61%	68.57%
gcc-60116	75,224	247	63,628	6,595	168	39,262	4,519	139	90,665	12,259	-43.72%	-17.26%
gcc-61383	32,449	195	32,527	10,534	84	23,169	3,302	117	49,259	16,318	-40.00%	39.29%
gcc-61917	85,359	103	10,047	1,359	65	23,138	2,953	58	14,477	2,144	-43.69%	-10.77%
gcc-64990	148,931	203	14,329	4,291	65	21,632	4,243	162	39,802	6,799	-20.20%	149.23%
gcc-65383	43,942	84	5,159	1,504	51	14,896	2,096	74	13,935	2,402	-11.90%	45.10%
gcc-66186	47,481	226	27,095	16,978	115	18,118	4,623	106	28,812	15,254	-53.10%	-7.83%
gcc-66375	65,488	227	28,829	11,301	56	21,181	7,036	82	31,952	14,695	-63.88%	46.43%
gcc-70127	154,816	230	29,949	14,762	84	21,507	5,856	87	27,903	10,481	-62.17%	3.57%
gcc-70586	212,259	94	15,135	16,918	130	35,710	5,607	36	25,617	12,455	-61.70%	-72.31%
gcc-71626	6,133	38	1,728	67	46	7,179	256	37	5,773	185	-2.63%	-19.57%
Median	70,356	106	14,294	4,263	72	21,344	3,296	83	26,760	6,682	-31.82%	-3.87%
Mean	94,487	151	21,124	6,174	85	21,999	3,563	89	33,654	7,789	-33.77%	12.66%

	Vulcan	ddSMT	Latra
Mean	121	109	103
Median	121	102	95
STD	68	66	61

(a) Reduction size

	Vulcan	ddSMT
Mean	-9.17%	-0.02%
Median	-2.86%	0.00%
STD	29.83%	32.82%

(b) C(%) w.r.t. baseline

	Vulcan	ddSMT	Latra
Mean	23708	2600	26048
Median	20418	2456	20294
STD	22099	2097	25108

(c) Property Queries

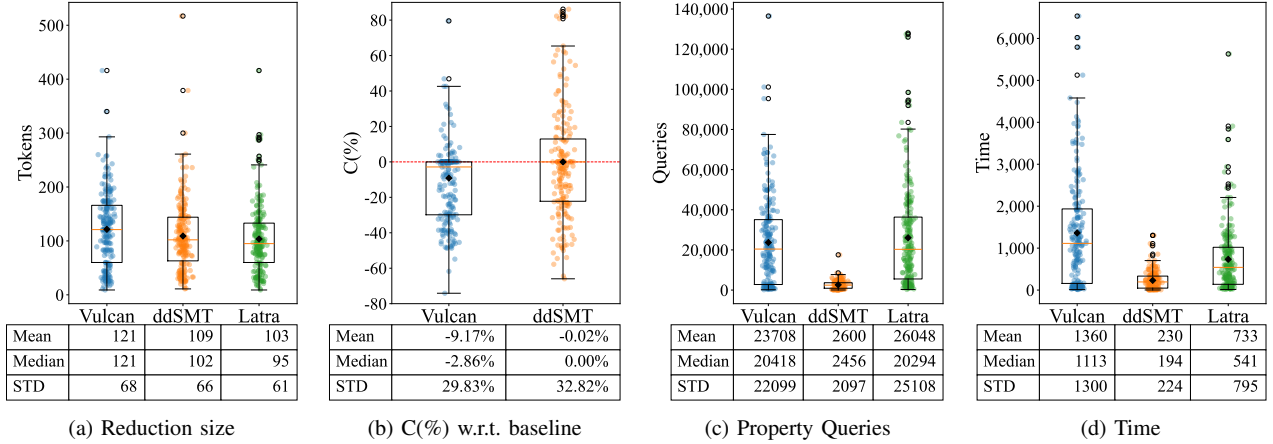|	Vulcan	ddSMT	Latra
Mean	1360	230	733
Median	1113	194	541
STD	1300	224	795

(d) Time

Fig. 4: The reduction results of Vulcan, ddSMT, and Latra after running on the SMT-LIB benchmarks.

process on SMT-LIB benchmarks. Table II shows that it takes Latra an average of 33,654 queries and 7,789 seconds to reduce the C programs, while Vulcan uses 21,124 queries and 6,174 seconds. However, when tested on SMT-LIB benchmarks, Latra is 32.27% faster than Vulcan. Figure 4c and Figure 4d show the distributions of the number of queries and time of each technique. Latra has successfully expedited the reduction, because Latra's SMT-LIB-specific transformations are able to remove more tokens in earlier iterations and allow the iterative reduction framework to converge more quickly and shorten the overall runtime. Due to the extra

transformation rules Latra has, more queries can be issued. Vulcan issues 23,708 while Latra issues 26,048 queries. With effective language-specific transformation, Latra is faster and more straightforward to identify a match and transform the code snippets than Vulcan.

Comparison with SPRs. Latra takes relatively longer time to complete the reduction process compared to C-Reduce and ddSMT. On average, C-Reduce uses 21,999 property tests and 3,563 seconds to reduce the token size from 94,487 to 85. Latra takes 33,654 queries and 7,789 seconds. As delineated in Figure 4d, when Latra operates on the SMT-LIB benchmarks,

it takes an average of 733 seconds while ddSMT finishes the process using an average of 230 seconds. We believe that the longer reduction time is primarily due to the fact that not all transformations implemented in C-Reduce and ddSMT were incorporated in our framework as a result of the limitations (see § III-E). Consequently, Vulcan in our framework would have a larger search space to identify matches, leading to increased reduction time.

> **RQ2**: In terms of reduction efficiency, Latra outperforms Vulcan in SMT-LIB benchmarks but is slower in C benchmarks. Among SPRs, Latra is relatively slower than C-Reduce and ddSMT.

D. RQ3: Engineering Effort

We used LOC as a proxy metric to evaluate the engineering effort of transformation rules from the *users' perspectives*.[1] Specifically, we manually reviewed the publicly available Git repository of both C-Reduce [26] and ddSMT [27] to compare the LOC required to implement their transformations. The results are displayed in Table I. For the C transformations, we compare our tool against the SPR C-Reduce. For the SMT-LIB transformations, we compare it against ddSMT.

Comparison with C-Reduce. As shown in the table, Latra significantly reduces LOC required to define transformations compared to C-Reduce. Comparing only the 27 transformations that Latra is feasible to implement, C-Reduce requires 5,685 LOC but Latra only requires 167 lines of rule definitions using our DSL. The lightweight transformation approach enables users to construct new transformations efficiently without learning and realizing the node traversal and modification logic manually. The concrete transformation process is fully automated in Latra. Hence, instead of writing hundreds of lines for each transformation, users can achieve the same by defining rules in just a few lines in Latra. This significantly reduces the engineering effort from the users' perspective.

Comparison with ddSMT. Compared to ddSMT, Latra typically requires fewer or the same number of LOC. As summarized in Table I, ddSMT implements transformations using 118 lines in total, whereas Latra only requires 62 lines. As discussed in § II-C, while ddSMT is less complex than C-Reduce, it still requires users to understand its tool infrastructure, such as utility functions, to implement new transformations. Additionally, the ddSMT developers have created over 1,200 lines of infrastructure code specifically for SMT-LIB program reduction, which cannot be applied and is unsuitable for other languages. In contrast, Latra provides

---

[1]While LOC provides a coarse but convenient proxy for engineering effort, it captures only a single dimension of the overall cost. A controlled user study could offer a more comprehensive and reliable evaluation. identifying an appropriate group of participants presents considerable challenges. Involving users with prior expertise in LLVM or Clang risks introducing bias against Latra, whereas involving participants without such expertise would likely make the tasks prohibitively time-consuming. Due to these methodological difficulties and the substantial resources required to design and conduct a rigorous user study, we defer this investigation to future work.

reusable infrastructure and a DSL that supports transformation development across multiple languages. This reduces the learning cost to implement SPRs for multiple languages and minimizes engineering effort.

> **RQ3**: In terms of LOC, Latra significantly reduces the engineering effort required for implementing program transformations compared to C-Reduce and ddSMT.

E. Threats to Validity

The threats to validity mainly lie in the implementation of Latra and the deployment of baselines. Incorrect implementation may lead to biased conclusions. To mitigate the threat, we carefully implemented the code, which is also reviewed by multiple authors. Besides, diverse test cases are designed to ensure Latra's implementation. As for the baseline reducers, we carefully followed their documentation when deploying them. Another potential threat is how representative the benchmarks used in the evaluation are. To improve the generalizability of the evaluation results, we utilized two popular benchmarks in our evaluation. These benchmarks cover two languages, and they have been widely used by related work [4], [18], [19], [24], [28]. The benchmarks selected for these experiments are derived from real-world bug reports, reflecting the capacity of Latra on real-world scenarios.

V. RELATED WORK

This section presents related studies for this work.

A. Program Reduction

AGRs. Delta Debugging (DD) [16] is the cornerstone for program reduction. The *minimizing delta debugging algorithm* (DDMin) treats a failure-inducing input as a flat sequence and iteratively partitions it such that the remaining subsequence can still trigger the failure. Several studies later improve the effectiveness and efficiency of DD from diverse perspectives [6], [7], [29]. To target specifically on inputs that have a tree structure, such as programs, Hierarchical Delta Debugging (HDD) algorithm [17] extends the concept by applying DDMin level-by-level starting from the root node of the tree. Hence, it can generate fewer syntactically invalid results. Later, to achieve full grammar-awareness, a syntax-guided program reduction technique, Perses [4], [30], is presented. Other than simply deleting nodes on the tree, Perses enables compatible node substitution. Furthermore, Vulcan [18] reduces the program beyond 1-minimality by introducing aggressive language-agnostic transformations during the reduction process, providing additional reduction opportunities. T-Rec [25] introduced a fine-grained language-agnostic reduction technique that leverages lexical syntax to explore the reduction opportunities in tokens. Tian et al. introduced a novel caching scheme to speed up program reduction [31]. Zhang et al. proposed the first technique that utilizes both the language generality of program reducers such as Perses and the language-specific

semantics learned by Large Language Models (LLMs) to perform language-specific program reduction [24].

SPRs. Designed for only specific languages, SPRs have the capability to leverage language-specific semantics. C-Reduce [11] is the state-of-the-art program reduction tool tailored for compiler bugs in C/C++ languages. Relying on Clang Libtooling [20], C-Reduce incorporates source-to-source code transformations to manipulate the programs. Another notable SPR is ddSMT [19]. It is specifically designed for SMT-LIBv2 language and its dialects. Similar to C-Reduce, ddSMT also adds simplification rules, e.g., replacing terms with default values of the same sort, in its framework to further simplify the inputs. Both tools have proven to improve the performance of program reduction in their targeted languages over AGRs due to the utilization of language-specific transformations. Latra combines both the advantages of the AGRs and the SPRs, applying user-defined language-specific transformation rules on the output of an AGR for potential smaller results.

B. Program Transformation

Heavyweight Imperative Approach. This approach emphasizes precise control over transformations. Tools such as Clang LibTooling [20] and OpenRewrite [32] usually operate on the AST-level of the program and provide frameworks for AST node traversal and tree manipulation. Therefore, it requires a deep semantic understanding of the target language in order to construct transformations programmatically using a heavyweight approach. On the other hand, this approach also enables a more fine-grained control. It would be suitable for transformations that require precise matching and modifications. Latra instead adopts a lightweight approach to eliminate the need for users to directly interact with node manipulation to implement transformations.

Lightweight Declarative Approach. This approach focuses on the ease of use and abstracts away the complexity of tree manipulation for implementing the transformations. Tools like ast-grep [33] and Coccinelle [34] allow specifying patterns in a declarative syntax. They are efficient for syntax-aware searching, replacing and refactoring. Other tools like Comby [22] and PolyglotPiranha [23] allow users to define transformation rules through templates. However, these tools often involve language grammar in template definitions. This makes them sensitive to updates in parsing frameworks, especially modifications in grammar. Moreover, template-based approaches may lead to the problem of over-generalization, which results in unintended modifications. Latra addresses these limitations by decoupling transformation rules from parser-specific grammars and introducing operation clauses in our DSL.

VI. Conclusion

This paper presents Latra, a template-based, language-agnostic transformation framework designed to enhance program reduction. By incorporating user-defined, language-specific transformations, Latra complements and refines reductions achieved by language-agnostic reducers such as

Perses and Vulcan. Using a straightforward DSL syntax, Latra enables users to define match-rewrite rules that identify and transform code without needing detailed knowledge of the target language's grammar. This approach allows Latra to combine the strengths of SPRs with the adaptability of AGRs, achieving both broad applicability and effective reduction results.

We evaluated Latra's performance across 20 benchmarks in C and 205 SMT benchmarks, demonstrating the versatility and effectiveness of the framework. Results show that Latra improves reduction results and speed, reducing 33.77% and 9.17% more tokens in C and SMT-LIB benchmarks than Vulcan, respectively. Additionally, Latra surpasses the language-specific reducer ddSMT with a higher token reduction rate across SMT-LIB benchmarks, and achieves comparable results with C-Reduce. These results highlight Latra's potential as a versatile framework for effective program reduction.

References

[1] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[2] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 386–399, 2015.

[3] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 347–361.

[4] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 361–371.

[5] T. L. Wang, Y. Tian, Y. Dong, Z. Xu, and C. Sun, "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*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 146–158. [Online]. Available: https://doi.org/10.1145/3575693.3575740

[6] M. Zhang, Z. Xu, Y. Tian, X. Cheng, and C. Sun, "Toward a better understanding of probabilistic delta debugging," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 2024–2035. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00117

[7] X. Zhou, Z. Xu, M. Zhang, Y. Tian, and C. Sun, "WDD: weighted delta debugging," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 1592–1603. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00071

[8] Y. Tian, Z. Xu, Y. Dong, C. Sun, and S. Cheung, "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, 2023, pp. 4873–4882. [Online]. Available: https://doi.org/10.24963/ijcai.2023/542

[9] Y. Xie, Z. Xu, Y. Tian, M. Zhou, X. Zhou, and C. Sun, "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*, M. Papadakis, M. B. Cohen, and P. Tonella, Eds. ACM, 2025, pp. 21–25. [Online]. Available: https://doi.org/10.1145/3713081.3731731

[10] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 294–305. [Online]. Available: https://doi.org/10.1145/2931037.2931074

[11] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[12] GCC-Wiki, "A guide to testcase reduction," 2020, retrieved 2024-08-22. [Online]. Available: https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[13] LLVM, "How to submit an llvm bug report," 2024, retrieved 2024-08-22. [Online]. Available: https://llvm.org/docs/HowToSubmitABug.html

[14] MozillaSecurity, "Using lithium," 2024, retrieved 2024-08-22. [Online]. Available: https://github.com/MozillaSecurity/lithium

[15] JerryScript, "Bug report," 2024, retrieved 2024-08-22. [Online]. Available: https://github.com/jerryscript-project/jerryscript/blob/master/.github/ISSUE_TEMPLATE/bug_report.md

[16] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[17] G. Misherghi and Z. Su, "Hdd: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 142–151. [Online]. Available: https://doi.org/10.1145/1134285.1134307

[18] Z. Xu, Y. Tian, M. Zhang, G. Zhao, Y. Jiang, and C. Sun, "Pushing the limit of 1-minimality of language-agnostic program reduction," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586049

[19] G. Kremer, A. Niemetz, and M. Preiner, "ddsmt 2.0: Better delta debugging for the smt-libv2 language and friends," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12760. Springer, 2021, pp. 231–242. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_11

[20] LLVM/Clang, "Clang documentation – libtooling," 2024, https://clang.llvm.org/docs/LibTooling.html, accessed: 2024-08-18.

[21] T. S.-L. Initiative, "Smt-lib benchmarks," Retrieved 2024-08-08 from https://smt-lib.org/benchmarks.shtml, 2023.

[22] R. van Tonder and C. Le Goues, "Lightweight multi-language syntax transformation with parser parser combinators," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 363–378. [Online]. Available: https://doi.org/10.1145/3314221.3314589

[23] A. Ketkar, D. Ramos, L. Clapp, R. Barik, and M. K. Ramanathan, "A lightweight polyglot code transformation language," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, jun 2024. [Online]. Available: https://doi.org/10.1145/3656429

[24] M. Zhang, Y. Tian, Z. Xu, Y. Dong, S. H. Tan, and C. Sun, "Lpr: Large language models-aided program reduction," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 261–273. [Online]. Available: https://doi.org/10.1145/3650212.3652126

[25] Z. Xu, Y. Tian, M. Zhang, J. Zhang, P. Liu, Y. Jiang, and C. Sun, "T-rec: Fine-grained language-agnostic program reduction guided by lexical syntax," *ACM Trans. Softw. Eng. Methodol.*, Aug. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3690631

[26] J. et al. Regehr, "C-reduce," Retrieved 2024-07-16 from https://github.com/csmith-project/creduce, 2012.

[27] G. Kremer, A. Niemetz, and M. Preiner, "ddsmt," Retrieved 2024-07-16 from https://github.com/ddsmt/ddSMT, 2021.

[28] M. Zhang, Z. Xu, Y. Tian, Y. Jiang, and C. Sun, "PPR: pairwise program reduction," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 338–349. [Online]. Available: https://doi.org/10.1145/3611643.3616275

[29] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 881–892. [Online]. Available: https://doi.org/10.1145/3468264.3468625

[30] J. L. Tian, M. Zhang, Z. Xu, Y. Tian, Y. Dong, and C. Sun, "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*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 2137–2141. [Online]. Available: https://doi.org/10.1145/3611643.3613101

[31] Y. Tian, X. Zhang, Y. Dong, Z. Xu, M. Zhang, Y. Jiang, S. Cheung, and C. Sun, "On the caching schemes to speed up program reduction," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, pp. 17:1–17:30, 2024. [Online]. Available: https://doi.org/10.1145/3617172

[32] Moderne, "Openrewrite," 2024, retrieved 2024-11-04. [Online]. Available: https://docs.openrewrite.org/

[33] Astgrep, "Ast-grep: Find code by syntax," 2024, retrieved 2024-11-4. [Online]. Available: https://ast-grep.github.io

[34] J. L. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," in *2018 USENIX Annual Technical Conference*, ser. USENIX ATC'18, Jul. 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/lawall