Large Language Models for Safe Minimization

***Abstract*—Several tasks in program analysis, verification, and testing are modeled as constraint solving problems, utilizing SMT solvers as the reasoning engine. In this work, we aim to investigate the reasoning capabilities of large language models (LLMs) toward reducing the size of an infeasible string constraint system by exploiting inter-constraint interactions such that the remaining ones are still unsatisfiable. We term this safe minimization.**

Motivated by preliminary observations of hallucination and error propagation in LLMs, we design SAFEMIN, a framework leveraging an LLM and SMT solver in tandem to ensure a safe and correct minimization. We test the applicability of our approach on string benchmarks from LeetCode in the computation of minimal unsatisfiable subsets (MUSes). We observed that SAFEMIN helps safely minimize 94.3% of these constraints, with an average minimization ratio of 98% relative to the MUSes. In addition, we assess SAFEMIN's capabilities in partially enumerating non-unique MUSes, which is baked into our approach via a "sample-and-enumerate" decoding strategy. Overall, we captured 42.1% more non-unique MUSes than without such LLM-based macro-reasoning. Finally, we demonstrate SAFEMIN's usefulness in detecting infeasible paths in programs.

I. INTRODUCTION

Several tasks in the domains of analysis, testing, and verification are modeled as constraint satisfaction problems. These include, but are not limited to, symbolic execution and automated test-case generation [1], [2], type checking, program verification [3], [4], [5], security [6], and optimization [7]. To establish these tasks, software developers express program states and program state transitions as logical formulae, and leverage Satisfiability Modulo Theories (SMT) solvers such as Z3 [8], CVC4/5 [9], [10], *etc.* as the back-end reasoning engines. For instance, in symbolic execution, each program path is represented as a logical constraint, aiding a systematic exploration of all possible execution paths. Similarly, in model checking, specifications of certain properties (*e.g.*, no `null` pointer is ever dereferenced) are expressed in a temporal logic, and the unreachability of the error state is verified.

Machine learning (ML) has been driving advancements to improve SMT's efficiency and scalability [11], [12], [13], particularly on enhancing and optimizing heuristic algorithms in solvers. ML-based solvers can be broadly classified into two categories: learning-aided solvers and stand-alone learning-based ones. *First*, learning-aided approaches combine ML techniques with modern solvers, replacing manually-crafted heuristics with more efficient ML-based ones [13], [14], [15], [16]. These include producing initialization assignments to achieve runtime reduction on local search [15], replacing searching heuristics with reinforcement learning to reduce the number of iterations [16], learning the distribution of the subset of clauses that is unsatisfiable (called UNSAT cores) to guide the solvers [14], [13], or learning the correlations among clauses by treating them as a non-ordered sequences with Transformers [17] to provide a learning-enhanced initialization heuristic for solvers. *Second*, stand-alone learning-based solvers predict satisfiability and decode value assignments with ML models. For instance, DeepSAT [18] formulates a solution as a product of conditional distributions and obtains the assignments by maximizing joint probabilities. Its assertions are based on probabilistic statistics. NeuroSAT [19] trains an GNN for satisfiability prediction.

With the recent emergent abilities of Large Language Models (LLMs), it is still unclear to what extent they can aid SMT solvers. While the state-of-the-art ML-based SMT solvers focus on deciding whether a constraint system is satisfiable (SAT) or unsatisfiable (UNSAT), in this work, we are interested in exploring the ability of LLMs in providing the evidential explanation of such a decision. In particular, we aim to investigate the LLM's ability in *safe minimization*, which is defined as follows: given an UNSAT constraint system, safe minimization reduces the size of the system such that the UNSAT property is preserved. In other words, safe minimization reduces the size of the constraint system from n to m ($m < n$) such that the remaining m constraints are still UNSAT. We refer to such subsets of the original constraint system as *smaller unsatisfiable subsets* (SUSes). A special case of SUSes is a minimal unsatisfiable subset (MUS). An MUS is defined as an unsatisfiable subset of a constraint system such that the removal of any constraint makes it satisfiable.

Safe minimization to SUSes or MUSes is crucial in several tasks. It could enhance the analysis of inconsistencies in a program, which may lead to unexpected behaviors. An UNSAT path constraint system contains multiple constraints representing the conditions in a program. An MUS is useful in localizing the inconsistencies among the conditions along the path due to its minimal size. These inconsistencies could stem from bugs, incorrect assumptions, conflicts among program components, or even design flaws and implementation errors. Localizing the source of such inconsistencies is crucial for software engineers to effectively tackle these challenges. Moreover, MUSes can be viewed as representing the *minimal reasons* or *explanations* of the infeasible paths.

In brief, we seek to answer the question: *"is it possible to safely minimize a constraint system, under the condition that no inconsistency-causing constraints are eliminated"*?

SMT solvers have specialized decision procedures spanning theories on integer and real number arithmetic, arrays, and strings. In this work, we focus on string constraints involving string operations such as concatenation, character and regex matching, string lengths, substrings, *etc*. The solvers on string theory are relevant for various tasks, including automated test-

case generation for scripting languages, static analysis of security vulnerabilities in web applications against code injection (caused by improper handling of untrusted strings), *etc*.

We conducted a preliminary empirical study in safe minimization with LLMs on multiple SMT-LIB string benchmarks extracted from LeetCode interview questions. We observed that LLMs are capable of *analyzing dependencies among the constraints* and can decompose an unsatisfiable set of constraints into sub-groups and reason over them (let us call this *macro-reasoning*) to discover the redundant constraints that can safely be eliminated. However, one key issue with them is their hallucinations. We reported two classes of hallucinations in the LLM-generated responses: (i) *orthographic*, where the constraints in generated candidate SUSes do not belong to the original constraint system, (ii) *rational*, where the reasoning used to produce the solution(s) or the candidate SUSes is flawed, due to which it is incorrectly assumed to be (un)satisfiable. With GPT-3.5, we reported parsing errors in 13.1% of the cases, *i.e.*, the model did not adhere to the prescribed output format due to which the SUS could not be retrieved. In 52.6% of the cases, it determined satisfiable constraint subsets as unsatisfiable. For the other LLMs, *e.g.,* GPT-4, Gemini-1.5 Pro, and Claude-3.5 Sonnet, there were no parsing errors. The number of cases where the model incorrectly determined satisfiable constraint subsets as unsatisfiable dropped to 18.6%, 19.3%, and 12%, respectively.

Accordingly, we incorporated the following strategies to mitigate hallucinations. First, for orthographic hallucinations, we employed a parser which uses edit distance as a metric to map the constraint generated by the LLM to the closest one in the input formula. Second, we utilized an SMT solver to verify the unsatisfiability of the candidate SUSes. Thus, we ensure that the candidate SUSes are both *correct* and *unsatisfiable*.

As an application for safe minimization, we used the SUSes produced by the LLM in combination with SMT solver as a verifier, and input them to the SMT solver, CVC5 [10], to compute the MUSes. This process of computing MUSes is more efficient due to a reduction in search space of MUSes from $O(2^n)$ to $O(2^m)$ $(m < n)$. In terms of finding MUSes in their constraint space, we can consider this as a *soft-search*, which allows for the LLM to not be penalized for imprecision.

Another dimension of complexity in MUSes stems from the fact that there can exist multiple non-unique ones for a given over-constrained system. However, enumerating all MUSes is typically intractable with respect to completion, and a formula with n constraints can have an order of 2^n MUSes. As a result, traditional MUS enumeration algorithms [20], [21], [22] do not complete enumeration within a reasonable time. More recently, there has been a shift towards avoiding such an exhaustive enumeration and computing an approximate count of the MUSes [23], [24], [25]. We bake such a parallelized, partial MUS enumeration into our approach by adopting a "*sample-and-enumerate*" decoding strategy within the LLM. We refer to this process as *self-exploration*, and the model as *Explorer LLM*. The intuition here, is that the Explorer LLM samples combinations of different macro-reasoning paths while analyzing the inter-constraint relationships, where each path is focused on explaining different sources of unsatisfiability and potentially leads to a distinct SUS.

Putting together the above ideas to enhance the capability of the LLMs, we propose SAFEMIN, a learning-aided solving framework that employs an LLM and an SMT solver in tandem to safely minimize a constraint system into its unsatisfiable subsets (SUSes), such that it encompasses all inconsistency-causing constraints. From our extensive evaluation of SAFEMIN framework on SMT-LIB string benchmarks, we were able to safely minimize 94.3% of the string constraints to their SUSes, with an average reduction of 98% relative to their corresponding MUSes, across all minimized ones. By adapting the MARCO algorithm [21], which is designed to enumerate minimal unsatisfiable subsets (MUSes) by searching the constraint space, we explored the computation of smaller unsatisfiable subsets (SUSes). Our results show that SAFEMIN significantly outperforms both random selection and search-based (breadth-first and depth-first) strategies in terms of minimization accuracy and minimization ratio. We include several stratified analyses probing the effect of the number of constraints in the formula on SAFEMIN's effectiveness in minimizing it. We compared SAFEMIN when being used with an SMT solver for MUS computation against using a traditional SMT solver directly, toward partial MUS enumeration, and noticed that our tool captures 42.1% more non-unique MUSes. Moreover, we conducted an in-depth analysis of the LLMs' macro-reasoning capabilities in SAFEMIN, revealing their understanding of decision procedures in SMT theories, as well as indicating its potential to scale to complex systems. Finally, we conducted a case study on the usability of our framework in identifying the conflicting conditions toward detecting infeasible paths in a real-world program. In brief, the key contributions of this paper include:

(1) To the best of our knowledge, this is the first study to explore the reasoning capabilities of LLMs on complex constraint benchmarks, such as SMT-COMP.
(2) SAFEMIN, a novel macro-reasoning approach for safely minimizing the constraint systems towards optimizing constraint solving. All our data is publicly available [26].
(3) SAFEMIN is a useful tool for computing MUSes, while also facilitating their parallelized, partial enumeration.

II. CONCEPTS AND ILLUSTRATIONS

A. Infeasible Constraint Systems

In Fig. 1, we present the conjunctive normal form (CNF) of a string formula extracted from the source code for an interview question related to the *partition* problem LeetCode [27], as documented in the SMT-LIB benchmarks [28]. This constraint system is unsatisfiable, and the constraint set $\{C_1, C_5, C_{19}, C_{21}\}$ (highlighted in orange) indicates one of its *minimal unsatisfiable subsets* (MUSes). The inconsistency is related to the string s, in which constraints C_1, C_5, and C_{19} assert that the characters at indices 1, 3, 5, and 6 are all the same, and C_{21} asserts that those at indices 3 and 6 can not be

```
1  At(s, 1) == At(s, 5)
2  Not(At(s, 1) == At(s, 4))
3  Length(s) <= 8
4  Length(s) == 8
5  At(s, 5) == At(s, 6)
6  At(s, 4) == At(s, 7)
7  Not(At(s, 4) == At(s, 6))
8  Not(At(s, 4) == At(s, 5))
9  Not(At(s, 5) == At(s, 7))
10 Not(Length(s) <= 7)
11 Not(Length(s) == 7)
12 Not(At(s, 6) == At(s, 7))
13 Not(Length(s) <= 6)
14 Not(Length(s) == 6)
15 Not(Length(s) <= 5)
16 Not(Length(s) == 5)
17 Not(Length(s) <= 4)
18 Not(Length(s) == 4)
19 At(s, 1) == At(s, 3)
20 Not(At(s, 3) == At(s, 7))
21 Not(At(s, 3) == At(s, 6))
22 Not(At(s, 3) == At(s, 5))
23 Not(At(s, 3) == At(s, 4))
24 Not(Length(s) <= 3)
25 Not(Length(s) == 3)
26 At(s, 1) == At(s, 2)
27 Not(At(s, 2) == At(s, 7))
28 Not(At(s, 2) == At(s, 6))
29 Not(At(s, 2) == At(s, 5))
30 Not(At(s, 2) == At(s, 4))
31 At(s, 2) == At(s, 3)
32 Not(Length(s) <= 2)
33 Not(Length(s) == 2)
34 Not(Length(s) <= 1)
35 Not(Length(s) == 1)
36 Not(Length(s) <= 0)
37 Not(Length(s) == 0)
```

Fig. 1. A motivating example

the same. Note: Removing any of the constraints in the MUS will no longer preserve such a contradiction (thus, *minimal*).

For a constraint system $C = \{C_1, C_2, ..., C_n\}$ over a set of variables, if we can find an assignment to all variables such that each C_i is satisfiable, *i.e.*, all restrictions of every C_i are met by the corresponding assignments, we say that C is *satisfiable*. Otherwise, it is considered to be *unsatisfiable*, or *infeasible*. In this work, we focus on analyzing infeasible string constraint systems on the basis of the following concepts:

Definition 1 (Smaller unsatisfiable subset (SUS))**.** *A smaller unsatisfiable subset of an unsatisfiable constraint system C is a subset $S \subseteq C$ such that S is still unsatisfiable.*

Definition 2 (Safe Minimization)**.** *Given an unsatisfiable constraint system C, the process of identifying a subset S such that it is still unsatisfiable (i.e., S is an SUS) is referred to as safe minimization. On the contrary, if S is satisfiable, it is unsafe minimization.*

Definition 3 (Minimal unsatisfiable subset (MUS))**.** *A minimal unsatisfiable subset of an unsatisfiable constraint system C is a subset $M \subseteq C$ such that M is still unsatisfiable and $\forall C_j \in M$, $M \backslash \{C_j\}$ is satisfiable. Note that the minimality in MUS refers to a set minimality, and not to minimum cardinality.*

Let us illustrate safe minimization via this example. For achieving safe minimization of the above UNSAT constraint system, consider constraint C_4 in Listing 1 which asserts that the length of a string `s` should be equal to 8. Next, consider pairs of constraints $\{C_{10}, C_{11}\}$, $\{C_{13}, C_{14}\}$, $\{C_{15}, C_{16}\}$, $\{C_{17}, C_{18}\}$, $\{C_{24}, C_{25}\}$, $\{C_{32}, C_{33}\}$, $\{C_{34}, C_{35}\}$, and $\{C_{36}, C_{37}\}$. Each of these resolves to `Length(s) > 7`, `Length(s) > 6`, `Length(s) > 5`, `Length(s) > 4`, `Length(s) > 3`, `Length(s) > 2`, `Length(s) > 1`, and `Length(s) > 0`, respectively. Furthermore, C_3 resolves to `Length(s) == 8` in conjunction with C_4. Thus, including C_4 and eliminating the constraints C_3, $C_{10}-C_{11}$, $C_{13}-C_{18}$, $C_{24}-C_{25}$, and $C_{32}-C_{37}$ would not affect the unsatisfiability of the resulting set.

Consider constraint C_1 which asserts that the character at index 1 in string `s` is the same as that at index 5. Accordingly, constraint C_2, which originally asserts that the character at index 1 in string `s` is not the same as at index 4, can be interpreted as the character at index 5 in string `s` not being the same as that at index 4. We can see that this is the same as constraint C_8, which is repetitive. Thus, C_8 can be eliminated without affecting the unsatisfiability of the resulting set. Using a similar analogy between constraint sets $\{C_5, C_9, C_{12}\}$, $\{C_5, C_{21}, C_{22}\}$, $\{C_5, C_{28}, C_{29}\}$, $\{C_6, C_{20}, C_{23}\}$, $\{C_6, C_{27}, C_{30}\}$, and $\{C_{21}, C_{31}, C_{28}\}$, the constraints C_{12}, C_{22}, C_{29}, C_{23}, C_{30}, and C_{28}, respectively, can safely be eliminated as well.

We refer to such a reasoning on sub-groups of constraints in a constraint system as *macro-reasoning*. We hypothesize that by exploiting the *inter-constraint relationships* via macro-reasoning, we can identify redundant constraints from the input formula while preserving the inconsistency in the SUS.

Definition 4 (Macro-Reasoning)**.** *Macro-Reasoning is the reasoning on sub-groups of an UNSAT constraint system that exploits the inter-constraint relationships to identify and eliminate redundant constraints and enable safe minimization.*

Recently, large language models (LLMs) have shown emergent behaviors with the abilities to reason in various domains, e.g., arithmetic [29], formal logic [29], [30], source code [31], [32], *etc*. Moreover, they have been shown to effectively produce proof steps towards proof generation [33]. This work *aims to evaluate whether LLMs can be leveraged to decompose a constraint system and macro-reason on groups of constraints via tractable steps of thoughts towards identifying (inferred) redundant constraints*. That is, we model this as a *soft-search* problem, to see whether the LLM can be leveraged to explore the constraint space to approximately identify the constraints that do not contribute to an inconsistency – the remaining which, can be encapsulated as *the SUS*. This contrasts with traditional exhaustive search on the entire constraint space [34].

For illustration, Fig. 2 displays the Hasse diagram for the power set lattice for a set of constraints $C = \{1, 2, 3, 4\}$. Edges represent containment relations. Starting from $\{1, 2, 3, 4\}$, the BFS ($\rightarrow$) and DFS ($\rightarrow$) strategies exhaustively explore the state space by eliminating constraints from C and checking the satisfiability of each subset. In contrast, SAFEMIN reasons about the *inter-constraint relations* to get to $\{2, 4\}$ (*i.e.*, an SUS) without exhaustive search, which helps converge faster to $\{2\}$ (*i.e.*, MUS). Our evaluation results empirically demonstrate this macro-reasoning capability of LLMs (Section IV).

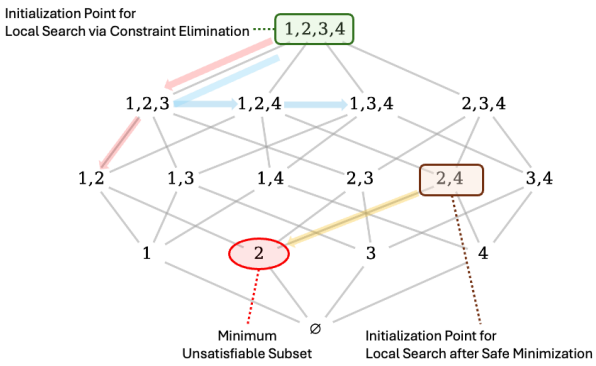

Fig. 2. Hasse diagram of the power set lattice for a generic set of four constraints $C = \{1, 2, 3, 4\}$. Starting from $\{1, 2, 3, 4\}$, one can explore the constraint space via local search strategies like depth-first ($\rightarrow$) and breadth-first ($\rightarrow$) search; or a macro-reasoning driven approach as in SAFEMIN.

B. Halluciation Mitigation

We conducted a preliminary experiment to study the capability of LLMs in macro-reasoning for safe minimization. We report our results in Section IV. A key limitation of large language models is that they can hallucinate, *i.e.*, generate plausible, but non-sensical information. This is concerning, as the generated candidate SUSes can not be trusted to be correct, or even unsatisfiable. From our empirical results, we report two classes of hallucinations in the LLM-generated response and leveraged the following strategies for their mitigation:

(i) *orthographic* hallucinations: the constraints in candidate SUSes do not belong to the original constraint system. For this type of hallucinations, a parser is used in which we map the generated constraint by the LLM to the closest one in the input formula via edit distances.

(ii) *rational* hallucinations: the reasoning used to produce candidate SUSes is incorrect, due to which satisfiable ones are marked unsatisfiable, and vice-versa. For the rational hallucinations, an SMT solver is used as a verifier to validate the LLM-generated candidates, and accept or reject them.

C. Parallelized Partial Enumeration of Multiple MUSes by Exploring Diverse Reasoning Paths

While computing MUSes is an application of SUSes produced by our framework, a constraint system might have multiple MUSes. An MUS for the above constraint system is highlighted in Listing 1. However, this string formula has multiple MUSes, including constraint sets $\{C_1, C_5, C_{26}, C_{28}\}$, $\{C_1, C_{26}, C_{29}\}$, $\{C_1, C_5, C_{21}, C_{26}, C_{31}\}$, *etc*. Finding *all* MUSes is typically intractable with respect to completion, and a formula with n constraints can have an order of 2^n MUSes. Thus, applications of MUSes tend to relax the completeness criterion by not focusing on finding all of them, but depend on the number produced within a certain time limit [23].

Complex reasoning tasks often have multiple reasoning paths. Wang *et al.* [35] make use of this diversity via a "sample-and-marginalize" decoding strategy in the LLM, where the optimal answer for a question is decided by marginalizing out the sampled reasoning paths and finding the most consistent one. Due to the nature of our task of minimizing constraint systems, we formulate it as a *soft search* problem. Thus, we propose a contrasting "sample-and-enumerate" decoding strategy. Here, the sampled reasoning paths can lead to multiple SUS candidates pertaining to non-unique MUSes. Such a design establishes a partial enumeration of MUSes. We call this strategy **self-exploration**, which encompasses the varied macro-reasoning perspectives of the LLM within a specific constraint system.

Theoretically speaking, increasing the size of the sample during self-exploration results in a more complete enumeration of MUSes. Furthermore, combining all such candidate SUSes with independent SMT-verifiers (as described in Section II-B) helps parallelize partial MUS enumeration with LLMs.

III. MACRO REASONING FOR SAFE MINIMIZATION OF UNSAT CONSTRAINT SYSTEMS

Our empirical results (Section V) showed that GPT-3.5 and GPT-4 exhibit macro reasoning capabilities in safe minimization of UNSAT constraint systems. However, they are still limited by halluciations and partial enumeration of multiple MUSes. Putting together the above ideas to enhance the macro-reasoning capabilitiess of LLMs in safe minimization, we propose SAFEMIN, a learning-aided solving framework that employs an LLM and an SMT solver in tandem, to safely minimize a constraint system into the unsatisfiable subsets (SUSes). This section presents SAFEMIN framework (Fig. 3).

An infeasible constraint system $C = \{C_1, C_2, ..., C_n\}$ often contains constraints, some of them contributing to the unsatisfiability of the system, and others that do not. Safe minimization aims to find the subsets of C, which are both *unsatisfiable*, and have *fewer* constraints than the input formulae. These subsets are referred to as the smaller unsatisfiable subsets (SUSes, $\mathcal{S}$). An SUS contains all constraints that lead to unsatisfiability in an MUS, possibly also including a few that do not. We posit that this approximation is desirable, as the *imprecision* provides *better correctness guarantees* in the form of unsatisfiability of the candidates.

SAFEMIN is designed to have the following key phases:

A. Self-Exploration

The goal, here, is to explore the constraint space of a given constraint system and identify candidate SUSes. Accordingly, we call this process *self-exploration*, and enable it by leveraging large language models (LLMs) along two dimensions:

1) Large Language Model as a Macro-Reasoning Agent: Inter-constraint relationships within MUSes encapsulate the underlying conflicts that lead to the unsatisfiability of a given constraint system. The primary role that the LLM serves is to capture such constraint interactions and decompose the constraint system into sub-groups of constraints. We surmize that reasoning on such sub-groups facilitates the identification of the root causes for inconsistencies, and subsequently, the SUSes. Furthermore, this process establishes tractable steps of thoughts for minimizing the constraint space.

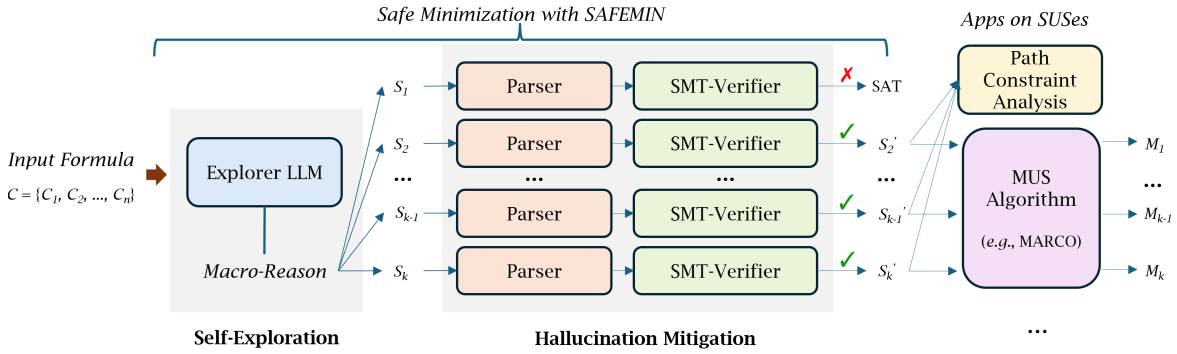

Fig. 3. Overview of SAFEMIN Framework and Its Applications

You are playing the role of an SMT solver on strings.

Instructions

Given a list of constraints or clauses, your goal is to identify a subset of these such that it is still unsatisfiable, i.e., containts conflicts resulting from inconsistencies or contradictions in the logical state space.

Follow these thought steps to find the conflicts or contradictions:
Step 1: Read and understand the given logical string formula to identify the variables, operators, and logical connectives used in the formula.
Step 2: Analyze dependencies both within, and between constraints. These can arise from string operations such as concatenation, length, position, substring extraction, etc., as well as the variables they share.
Step 3: Macro-reason on these constraints to find pairs or groups that can possibly be combined and resolved. It is okay for there to be overlap among the different groups of constraints. The motivation for this grouping, however, should be to find redundant constraints in the input, which can be inferred from combining parts of these groups. Note that such redundant constraints CAN be eliminated.
Step 4: Next, try to identify constraints or groups of constraints which also satisfy another constraint or group of constraints. In such a case, the latter constraint or group CAN be eliminated.
For steps 3 and 4, try to find as many such groups as possible.
Step 5: Try to identify pairs of constraints that directly conflict with each other. These need to definitely be part of the output subset. Note that such conflicts or inconsistencies WILL arise in the given example. Think step by step.

Output Format

After identifying all such constraints that can safely be removed, output a comma-separated list of all remaining clauses. Each clause should be inserted between <c> and </c> tags, and the output between <output> and </output> tags.

Disclaimer

Try to minimize the input constraints as best as possible, while also ensuring the output subset's unsatisfiability.

Fig. 4. System prompt for Explorer LLM in SAFEMIN.

In Fig. 4, we present the prompt given to the LLM for this purpose, which primarily illustrates the *Instructions* for it to pursue a Chain-of-Thought (CoT) [36] reasoning and decompose the input constraint system into constraint subgroups, to macro-reason about them; and the prescribed *Output Format* for the candidate SUSes. In our preliminary experiments, we observed that directing the LLM to output the entire list of constraints that belong to the candidate SUSes nudges it to recall the inferred knowledge from the macro-reasoning steps, as opposed to using special markers for identifying the constraints. In this way, the LLM plays the role of exploring the constraint space towards summarizing the conflicts in a constraint system in the form of their SUSes.

2) Exploring Diverse Reasoning Paths: As noted earlier, it is possible for a constraint system to have multiple MUSes, *i.e.*, multiple sources of inconsistencies. In this context, another dimension of LLM operability is the *parallelized, partial enumeration of such non-unique MUSes*.

To enable this process, we propose a *"sample-and-enumerate"* decoding procedure, where we first *sample* from the LLM's decoder to generate a diverse set of reasoning paths. While the basis of these paths is rooted in *macro-reasoning on the input formula* (as in Section III-A1), each might be focused on a different source of inconsistency – thus leading the LLM to *explore* non-unique candidate SUSes. Subsequently, we can leverage these candidates to extract non-unique MUSes.

While self-exploration is similar in spirit to *self-consistency* decoding strategy [35], a notable difference is that the latter follows the "sample-and-marginalize" approach, wherein, marginalization is used to converge to the most consistent response. As our task is rooted in search space exploration, we do not aggregate the reasoning paths in this manner, but use the *divergence* to *enumerate multiple candidates* instead.

Another major advantage of this design is that it naturally facilitates parallelization, as the candidate SUSes can be independently analyzed towards computing the MUSes. In theory, sufficiently increasing the sample size during self-exploration can result in a complete enumeration of all MUSes.

B. Ensuring Correctness

Large Language Models (LLMs) can hallucinate, and the candidate SUSes generated by Explorer LLM can contain constraints that do not belong to the input constraint system, or they themselves can possibly not be unsatisfiable (as is required by the definition of an SUS). To mitigate hallucinations of this form, we incorporate two forms of correctness verification into SAFEMIN's design:

1) Constraint Validity: The first class of hallucinations is *orthographic*, as it is possible that a constraint in the candidate SUS could not belong to the input constraint set altogether. To mitigate this, we incorporate a *Parser* into our design, which maps the incorrect constraint to the closest constraint in the input formula using *edit distance* as a heuristic. We adopt this metric for such an "auto-correction" as it is lightweight. Note

that this can be swapped with *embedding-based similarity metrics* as well, which is left for future work. Overall, this step ensures the validity of the candidate SUSes.

2) Unsatisfiability Verification: The next class of hallucinations is *rationale*-specific, as it is possible for the LLM-generated candidate SUSes, which it deduces to be unsatisfiable via several macro-reasoning steps, is not actually so.

In this case, we see two ways of ensuring correctness, by using an *SMT solver* to verify (a) the correctness of the macro-reasoning steps, (b) the unsatisfiability of the final generated candidate SUS. While the former ensures a robust generation where only valid macro-reasoning steps are considered to infer the SUS, it can require multiple calls to the solver and can be prohibitively expensive. Moreover, during our preliminary evaluation, we also observed that the LLM sometimes tends to self-correct through the course of generation. Thus, we chose to include an SMT verifier after the fact, *i.e.*, which checks whether the final generated candidate SUS is indeed unsatisfiable – ensuring the correctness of candidate SUSes.

Overall, the *Explorer LLM* (for self-exploration, as in Section III-A), *Parser* and *SMT Verifier* (for correctness verification, as in Section III-B) work in tandem to ensure the safe minimization of unsatisfiable constraint systems, enabling a parallelized enumeration of SUSes and MUSes later.

C. Mathematical Formulation

For an unsatisfiable input constraint system $C = \{C_1, C_2, ..., C_n\}$, let $\mathcal{P}\{C\}$ be its power set, *i.e.*, $\mathcal{P}(C) = \{\phi, \{C_1\}, \{C_2\}, ..., \{C_n\}, \{C_1, C_2\}, ..., \{C_1, C_2, ..., C_n\}\}$. The Explorer LLM in *self-exploration* phase takes C as input, and returns multiple candidate SUSes $\cup_{i=1}^{k} S_i$, where $S_i \in \mathcal{P}(C)$ and k is its decoding sample size.

Next, we check $\cup_{i=1}^{k} S_i$ for *correctness* to obtain $\cup_{i=1}^{l} S_i$ ($l \leq k$) such that, $S_i \in \mathcal{P}(C)$ (*i.e.*, from Parser) and S_i is indeed unsatisfiable (*i.e.*, from SMT-Verifier). Finally, MUSes can be extracted in parallel from $\cup_{i=1}^{l} S_i$.

IV. Empirical Evaluation

We conducted several experiments, seeking to answer the following research questions:

(I) Intrinsic Evaluation

RQ$_1$. Effectiveness in Safe Minimization of String Constraints: (i) *Can the LLMs in* SafeMin *exploit inter-constraint relations toward safe minimization?*, and (ii) *How well do they safely minimize string constraint systems?*

(II) Qualitative Analysis of Macro-Reasoning

RQ$_2$. *How accurate is the LLM's macro-reasoning on input string constraint systems toward safe minimization?*

(III) Extrinsic Evaluation

RQ$_3$. Safe Minimization toward MUS Computation: *How close are the SUSes produced by* SafeMin *to the actual MUSes?*

RQ$_4$. Diverse Reasoning Paths for SUSes toward MUSes: *How useful is* SafeMin *in using diverse reasoning paths in enumerating different MUSes for a constraint system?*

TABLE I
Data Statistics.

#-Constraints	#-Instances	#-Operations
0 – 10	47	23.3
10 – 20	110	57.7
20 – 30	80	89.7
30 – 40	118	114.7
40 – 50	33	140.2
Overall	**388**	**84.5**

RQ$_5$. Detection of Infeasible Paths in Source Code: *How useful is* SafeMin *in detecting infeasible paths?*

V. Effectiveness in Safe Minimization

A. Data Collection

With the first objective being to evaluate the LLM's capabilities in safe minimization, we used the string benchmark set in SMT-COMP'23 [37] that contains quantifier-free string formulae with constraints reasoning about string lengths (*i.e.*, QF_SLIA from QF_Strings division). In particular, we picked the *LeetCode* benchmark containing 2,666 string formulae obtained from several LeetCode interview questions. These cover a wide range of complex string equations, inequalities, regular expressions, and include string functions such as `str.indexof`, `str.substr`, and `str.at`. In Table I, we list the total number of instances, and the mean number of string operations.

To build the ground truth, we leveraged the state-of-the-art tool CVC5 [10] to solve these constraints. Of the 2,666, we identified a total of 774 unsatisfiable instances. Since SafeMin takes unsatisfiable sets of constraints as input, we disregard the remaining 1,892 instances. Next, we simplified the unsatisfiable instances algebraically via rewriting operations to extract corresponding lists of constraints. Finally, we stratified them based on the number of constraints in each formula, and split them equally into *validation* and *test* sets. We use the former for tuning the prompt in Explorer LLM (Fig. 3), and report the final performance on test split.

B. Experiment Setup

1) Baselines: In this experiment, we aim to assess how well the LLMs in SafeMin reduce the given unsatisfiable set of constraints to the corresponding SUS toward safe minimization. First, we adopted Chain-of-Thought (CoT) [36] prompting to assess the models' macro-reasoning capabilities. Then, we used Chain-of-Thought prompting with Self-Exploration (CoT-$\mathcal{SE}$), enabling them to explore diverse reasoning paths and capture multiple SUSes. We compare the performance of SafeMin when using both prompting strategies, with GPT-3.5, GPT-4, Gemini-1.5 Pro, and Claude 3.5 Sonnet.

Consider a constraint system C=$\{C_1, C_2, ..., C_n\}$ which has a power set lattice $\mathcal{P}(C)$ = $\{\phi, \{C_1\}$, $\{C_2\}$,..., $\{C_n\}$, $\{C_1, C_2\}$, ..., $\{C_1, C_2, ..., C_n\}\}$. As the baselines for our advanced prompting in SafeMin, we considered state space exploration with systematic search-based, as well as random strategies. *First*, we compared the CoT-based approaches with a baseline that randomly selects an element from $\mathcal{P}(C)$ (*i.e.*,

TABLE II
EFFECTIVENESS EVALUATION ON STRING CONSTRAINTS.

#-Constraints (→) / Approach (↓)	Evaluation Metrics							
	0 – 10 ($\|D\| = 47$)	10 – 20 ($\|D\| = 110$)	20 – 30 ($\|D\| = 80$)	30 – 40 ($\|D\| = 118$)	40 – 50 ($\|D\| = 33$)	*Total* ($\|D\| = 388$)	$m_{SUS,\,\mathcal{D}_u}$	$m_{SUS,\,\mathcal{D}}$
Random (*pass*@1)	15	39	18	42	14	128 (32.9%)	0.46	0.15
CoT *w/* GPT-3.5	14	48	27	34	7	130 (33.5%)	0.41	0.14
GPT-4	41	91	69	85	24	310 (79.9%)	0.61	0.48
Gemini-1.5 Pro	41	90	58	85	23	297 (76.5%)	0.72	0.58
Claude-3.5 Sonnet	40	84	73	99	28	**324 (83.5%)**	**0.81**	**0.78**
Random (*pass*@5)	39	87	52	97	27	302 (77.8%)	0.45	0.35
Depth-First (*max_visited*=5)	29	90	69	109	31	328 (84.5%)	0.05	0.05
Breadth-First (*max_visited*=5)	31	104	76	114	33	358 (92.3%)	0.06	0.05
CoT-$\mathcal{SE}$ *w/* GPT-3.5	38	90	56	101	22	307 (79.1%)	0.48	0.38
GPT-4	44	107	77	115	29	**372 (95.9%)**	0.70	0.67
Gemini-1.5 Pro	42	92	59	96	27	316 (81.4%)	0.76	0.64
Claude-3.5 Sonnet	45	97	75	117	32	366 (94.3%)	**0.83**	**0.82**

random pass@1). These are equivalent, since both check the unsatisfiability of the candidate SUS once. *Second*, to benchmark our CoT-$\mathcal{SE}$, we also established a baseline that randomly selects k elements from $\mathcal{P}(C)$ (*i.e.*, random pass@k), where k is the number of samples produced in CoT-$\mathcal{SE}$.

Finally, adapting the traditional, non-LLM-based MARCO algorithm [21], which employs systematic state space exploration to enumerate MUSes, we also established baselines that eliminate the constraints in the input constraint set C by exploring $\mathcal{P}(C)$ via breadth-first (BFS) and depth-first search (DFS), checking the unsatisfiability of the subsets in order to enumerate the SUSes. For a fair comparison with CoT-$\mathcal{SE}$, which checks for unsatisfiability of k candidate SUSes, we limit the traversal in BFS and DFS to a maximum of k nodes, involving a maximum of k checks of unsatisfiability. In Fig. 2, we present an illustration of such search-based baselines. Also, given the computational cost of the unsatisfiability checks with an SMT solver, we set k as 5 in this work. In Table II, we call these baselines Depth-First and Breadth-First (*max_visited*=5).

2) Metrics: To intrinsically measure the models' performance in macro reasoning for safe minimization, we define *Minimization Accuracy* (A_m), which measures the ratio of the number of instances in which the candidate SUSes are unsatisfiable to the total number of instances. If D represents the test set and D_u represents the set of instances in which the candidate SUSes are indeed unsatisfiable, mathematically, $A_m = \frac{|D_u|}{|D|}$. In the case of Chain-of-Thought prompting with Self-Exploration (*i.e.*, CoT-$\mathcal{SE}$), where there are multiple candidate SUSes, we consider the prediction of even one unsatisfiable candidate SUS as a correct instance.

While A_m measures the proportion of instances in the dataset in which the input formula is safely minimized, it does not quantify the reduction in search space. Thus, we measure the quality of safe minimization of the constraint system C to the SUS S (*i.e.*, $C \rightarrow S$) with *Minimization Ratio*, which is defined as $m_{SUS} = \frac{|C|-|S|}{|C|}$. Finally, we define two aggregated variants of m: (a) that on the entire test set, *i.e.*, $m_{SUS,\,\mathcal{D}} = \frac{1}{|\mathcal{D}|}\sum_{\in\mathcal{D}} m_{SUS}$, and (b) that where candidate SUSes are unsatisfiable, $m_{SUS,\,\mathcal{D}_u} = \frac{1}{|\mathcal{D}_u|}\sum_{\in\mathcal{D}_u} m_{SUS}$.

C. Experiment Results

1) Comparative Results: In Table II, we report the performance of different variants of the models. Interestingly, when prompted *without self-exploration* from SAFEMIN, GPT-3.5 (row 2) performs slightly better than the naive approach (row 1), despite the latter only picking a random constraint subset from $\mathcal{P}(C)$ as the SUS. Upon further analyzing the failures, we discovered that there were parsing errors in 13.1% of the cases, *i.e.*, the model did not adhere to the prescribed output format due to which the SUS could not be retrieved. Moreover, in *52.6% of the cases, it determined satisfiable constraint subsets as unsatisfiable*, which we were able to reject with the SMT verifier. This shows the limitations of GPT-3.5 in following our instructions for computing the SUS.

In contrast, by prompting GPT-4 with the same instructions (*i.e.*, CoT, row 3), we observed an improvement in performance by 138.5%. This establishes a direct comparison between both GPT variants in their ability to follow our instructions and CoT steps. Such improvement is 128.5% for Gemini-1.5 (row 4) and 149.2% for Claude-3.5 (row 5). In particular, there were *no* parsing errors with these LLMs, and the number of cases in which the model incorrectly determined satisfiable constraint subsets as unsatisfiable dropped to 18.6%, 19.3%, and 12.4% (rows 3–5).

SAFEMIN aims to address these limitations and enhance the LLM's ability to safely minimize constraint sets. Compared to the baseline with 5 random selections, we observed the improvements for SAFEMIN with GPT-4, Gemini-1.5 Pro and Claude-3.5 Sonnet from 4.6%–23.2% in A_m and 8.6%–91.4% in m_{SUS}. This improvement for SAFEMIN with GPT-3.5 over that random picker is smaller as GPT-3.5 is limited as seen.

We also compared SAFEMIN with the MARCO algorithm with systematic BFS and DFS searches. With the traversal limited to a maximum of 5 nodes, eliminating constraints to navigate the state space results in a limited exploration depth. Accordingly, we observed *low minimization ratios* for both baselines. However, the possibility of hitting a candidate SUS which is a superset of the MUS in the constraint state subspace is high. As a result, the search-based baselines achieve a high

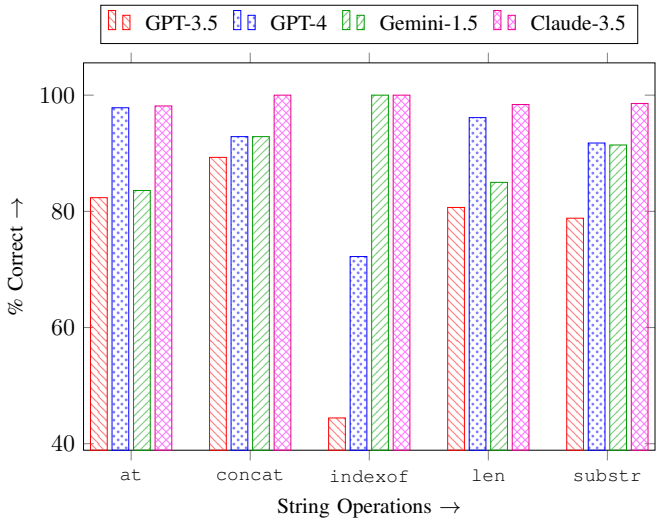

Fig. 5. Performance comparison of CoT-$\mathcal{SE}$ in LLMs on the most-occurring string operations: `at`, `concat`, `indexof`, `len`, and `substr`.

minimization accuracy. This trade-off shows the limitation of search-based strategies for safely minimizing constraint sets.

Next, we compare the performance of our *self-exploration* prompting with different LLMs (*i.e.*, rows 9–12). As seen, SAFEMIN improves the performance over only-CoT in minimization accuracy by 136.1%, 20%, 6.4%, and 13% when using GPT-3.5, GPT-4, Gemini-1.5 Pro, and Claude-3.5 Sonnet, respectively. Moreover, all LLMs with CoT-$\mathcal{SE}$ outperform their non-$\mathcal{SE}$ variants across all intervals, when stratified on the number of constraints, demonstrating the effectiveness of CoT-$\mathcal{SE}$ in SAFEMIN. Finally, while SAFEMIN with GPT-4 and Claude-3.5 Sonnet achieve similar minimization accuracy, the minimization ratio for the latter is 17.1% higher.

2) Performance on String Operations: Fig. 5 shows the LLMs' performance with CoT-$\mathcal{SE}$ prompting on Top-5 most occurring SMT2 string operations. Among the instances containing `str.at` (string-character function), `str.concat` (string concatenation function), `str.len` (string length function), and `str.substr` (substring function), all LLMs make correct predictions in more than 80% of the cases (with GPT-4 and Claude-3.5 showing a better understanding of these operations). GPT-3.5 records a particularly low performance for `str.indexof` (returns the index of the first occurrence of the specified value), minimizing only 21% of these occurrences correctly. This can be attributed to the complexity of the `str.indexof` function, which often occurred in more complex constraints with other operations, which GPT-3.5 failed to resolve. In contrast, GPT-4 correctly predicts 72.2% of these instances, while Gemini-1.5 and Claude-3.5's correctly predict all of them. This result exhibits the improved macro-reasoning capabilities of the other LLMs over GPT-3.5.

Overall, with SAFEMIN's CoT-$\mathcal{SE}$ prompting strategy, we record the best performance with GPT-4 (row 10) and Claude-3.5 Sonnet (row 12), *safely minimizing the input formulae to SUSes in* $\sim$ 95% *of the cases*. It outperforms the other baselines in minimization accuracy by 3.9%–190.6% and minimization ratio by 17.1%–1540%. In brief, we can see that SAFEMIN with GPT-4 or Claude-3.5 Sonnet, when prompted with CoT-$\mathcal{SE}$, was able to navigate the intricacies of the constraints and inter-constraint relations/inconsistencies.

> **RA$_1$.** (1) GPT-4 and Claude-3.5 Sonnet demonstrate superior macro-reasoning capabilities in exploring inter-constraint relationships to achieve safe minimization.
> (2) SAFEMIN with CoT-$\mathcal{SE}$ enhances the LLMs' macro-reasoning about sources of inconsistencies, enabling safe minimization of $\sim$**95%** of the string constraints.

3) Time Efficiency for Safe Minimization: We report the time for safe minimization taken by all LLMs for safely minimizing the input formulae when prompted with SAFEMIN's CoT-$\mathcal{SE}$ strategy. For GPT-3.5, the minimum observed processing time was 0.84 seconds, while the maximum recorded time was 142.3 seconds. The mean processing time for GPT-3.5 was 26.3 seconds (with a standard deviation of 31.8 seconds). Conversely, GPT-4 recorded minimum and maximum processing times of 1.13 and 111.3 seconds, respectively. The mean processing time for GPT-4 was 20.97 seconds (with a standard deviation of 25.87 seconds). For Claude-3.5 Sonnet, the times ranged from 0.64 seconds to 103.2 seconds, with a mean of 16.9 seconds and a standard deviation of 14.1 seconds. Similarly, Gemini-1.5 Pro recorded a minimum processing time of 0.72 seconds and a maximum of 122.6 seconds, with a mean of 22.1 seconds and a standard deviation of 21.2 seconds. While being subject to network latency, these metrics provide insights into scaling to real-world systems with LLMs.

VI. QUALITATIVE ANALYSIS OF MACRO-REASONING

In this experiment, we aim to assess the quality of macro-reasoning of LLMs used within SAFEMIN. Due to the effort involved in manually parsing through the LLM's textual responses, we randomly picked 5 examples from each of the input formulae sub-groups in Section V-A (*i.e.*, 0–10, 10–20, ..., 40–50), collecting a total of 25 instances and their corresponding LLM responses, when prompted with CoT-$\mathcal{SE}$.

From our analysis, we could overarchingly identify the following *strengths* in all LLMs' macro-reasoning:

(S_1) Able to pick contradicting logic pairs.
(S_2) Able to resolve intricate combinations of operations.
Other LLMs – not GPT-3.5:
(S_3) Able to combine constraints via transitive inference.
(S_4) Understands `str.substr` and `str.at` operations well.
(S_5) Able to combine and resolve combinations of operations, *e.g.*, the length of a substring (`str.len` and `str.substr`).

We observed the following categories of *inaccuracies* in all LLMs' macro-reasoning on constraint sub-groups:

(W_1) Makes some logical errors when constraints involve `not` operation. For *e.g.*, GPT-3.5 resolved `not(a≤b)` as `a≤b`.
(W_2) Some errors in combining constraints. For *e.g.*, GPT-3.5 resolved `(a==1) and not(a≤1)` as `not(a==1)`.
(W_3) Some errors in understanding complex constraints. For *e.g.*, `str.len(str.substr(s, 0, 1)) == 1`

TABLE III
QUALITATIVE STUDY ON GPT MODELS' MACRO-REASONING.

Weaknesses (→) LLM (↓)	W_1	W_2	W_3	W_4	W_5	W_6
GPT-3.5	5	4	5	7	4	2
GPT-4	4	1	1	–	–	–
Claude-3.5 Sonnet	2	2	1	–	3	–

Only GPT-3.5:

(W_4) Tends to forget some of the previously inferred steps.

(W_5) Combines constraints based on incorrect reasoning.

(W_6) Fails to localize sources of inconsistencies.

Table III displays the counts of LLMs for all inaccuracies listed above (except Gemini-1.5, which only produced the output constraint subsets without any analyses, due to which we could not identify the weaknesses). Note that these counts are not mutually exclusive, *i.e.*, one instance can be counted multiple times across categories. Furthermore, in some cases, we noticed that GPT-4 and Claude-3.5 tend to ignore the prescribed thought steps, even more so than GPT-3.5. However, its superior performance is indicative of its adaptability during constraint resolution. Refer to [26] for more details.

As an illustration, consider the string formula in Fig. 6. GPT-4, when prompted with CoT-$\mathcal{SE}$, was able to not only safely minimize it, but also accurately predict the corresponding MUS, *i.e.*, $\{C_1, C_4, C_6\}$. Notably, it was able to: (a) resolve constraints involving multiple variables (C_1 and C_6); (b) understand and analyze constraints involving intricate `Concat` and `str.substr` operations (C_3, C_4 and C_5).

We also tested the motivating example from Fig. 1 with GPT-4 using CoT-$\mathcal{SE}$. Interestingly, we observed that GPT-4 was able to: (a) group all constraints about `Length(s)` to combine them into a single constraint `Length(s) == 8`; (b) reason about the transitive relationships between C_1, C_{26}, and C_{29}, to recognize that C_1 and C_{26} together would contradict C_{29}, since `At(s, 1) == At(s, 5)` and `At(s, 1) == At(s, 2)` would resolve to `At(s, 2) == At(s, 5)`, which contradicts with its negation (C_{29}). These reasoning steps align with our hypotheses in Section II, thus confirming the LLMs' capabilities in reasoning-based safe minimization.

RA$_2$. LLMs demonstrate superior macro-reasoning capabilities to reason about the inter-constraint inconsistencies and safely minimize infeasible constraint systems with a high accuracy, thus exhibiting a potential to scale.

VII. SAFE MINIMIZATION TOWARD MUS COMPUTATION

In this study, we aim to explore the usefulness of SUSes predicted by SAFEMIN in the application of computing MUSes with the reduction of search space via safe minimization.

A. Data Collection

First, we leverage CVC5 to compute the MUSes for all constraint sets in the test set in Section V-A. At its core, CVC5 uses either the constructed proof, or an assumption-based approach for MUS extraction. Due to its lightweight nature, we opt for the latter. Accordingly, for each instance in the test set, we establish $\langle C, \mathcal{S}, \mathcal{M} \rangle$ tuples, where, each corresponds to the input constraint set, candidate SUS from SAFEMIN, and the corresponding MUS computed by cvc5, respectively. Note that we can extract multiple $\mathcal{M}$'s for a C. In this experiment, we ensure that both C and $\mathcal{S}$ correspond to the same $\mathcal{M}$.

```
1  Not(beginWord == endWord)
2  Not(Length(beginWord) <= 0)
3  endWord == Concat(str.substr(beginWord, 0, 2),
       Concat("t", str.substr(beginWord, 3, -3 +
       Length(beginWord))))
4  endWord == Concat(str.substr(beginWord, 0, 1),
       Concat("o", str.substr(beginWord, 2, -2 +
       Length(beginWord))))
5  endWord == Concat("d", str.substr(beginWord, 1, -1 +
       Length(beginWord)))
6  beginWord == "dot"
7  Length(beginWord) >= 3
8  Length(beginWord) >= 2
9  Length(beginWord) >= 1
```

Fig. 6. String formula safely minimized by GPT-4 with CoT-$\mathcal{SE}$ prompting.

B. Experiment Setup

1) Baselines: We compare all LLMs with the best-performing CoT-$\mathcal{SE}$ prompting for SAFEMIN (in Section V), as used for computing MUSes from the corresponding SUSes.

2) Metrics: While Section V measures the reduction in search space of the input formula, it does not measure this reduction relative to the size of the MUS. To this effect, we define *minimization ratio* as $m_{MUS} = \frac{|C|-|\mathcal{S}|}{|C|-|\mathcal{M}|}$. By definition, m_{MUS} ranges from 0 to 1. If $m_{MUS} \to 0$, it means that Explorer LLM in SAFEMIN removed few or no constraints from C, and $\mathcal{S} \approx C$. On the contrary, if $m_{MUS} \to 1$, it indicates that Explorer LLM removed most of the non conflict-causing constraints from C, *i.e.*, $\mathcal{S} \approx \mathcal{M}$. Similar to RQ1, we define two aggregated variants, *i.e.*, $m_{MUS,\,\mathcal{D}}$ and $m_{MUS,\,\mathcal{D}_u}$.

C. Experiment Results

In Table IV, we report the minimization ratios for all LLMs with SAFEMIN's CoT-$\mathcal{SE}$ prompting strategy. This illustrates the usefulness of the SUSes generated in Section V-C, in the context of their corresponding MUSes. Overall, we can see that among the 366 safely minimized SUSes (as noted in Table II), *Claude-3.5 Sonnet records an average minimization ratio with respect to their MUSes by* **98%**, followed by Gemini-1.5 Pro at 90%, GPT-4 at 82%, and GPT-3.5 at 57%. That is, the SUSes generated by Claude-3.5 Sonnet are on average only 2% larger than the corresponding MUSes. When also including the 22 instances for which Claude-3.5 Sonnet could not safely minimize to the SUS (here, m=0), the average is 96%. For Gemini-1.5 Pro and GPT-4, these aggregated minimization ratio variants are 76% and 79%, respectively.

The breakdown in Table II on the basis of the number of constraints further sheds light on the quality of the SUSes. For instance, consider the string formulae containing 40–50 constraints. GPT-3.5 safely minimizes 66.7% of them (as noted in Table II). The average minimization in these formulae is

TABLE IV
[USEFULNESS OF SUSES PRODUCED BY SAFEMIN] EFFECTIVENESS OF SAFE MINIMIZATION TOWARD MUS COMPUTATION.

#-Constraints (→) / Approach (↓)	Evaluation Metrics (*i.e.*, $m_{MUS,\mathcal{D}_i}$)											
	0 – 10		10 – 20		20 – 30		30 – 40		40 – 50		***Total***	
	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$	$\mathcal{D}_u$	$\mathcal{D}$
CoT-$\mathcal{SE}$ *w/* GPT-3.5	0.65	0.53	0.56	0.46	0.63	0.44	0.54	0.46	0.39	0.26	0.57	0.45
GPT-4	0.84	0.79	0.82	0.79	0.86	0.83	0.81	0.79	0.73	0.65	0.82	0.79
Gemini-1.5 Pro	0.88	0.81	0.88	0.81	0.92	0.72	0.90	0.73	0.89	0.73	0.90	0.76
Claude-3.5 Sonnet	0.98	0.96	0.96	0.93	0.98	0.97	0.98	0.98	0.99	0.96	**0.98**	**0.96**

39.4%, *i.e.*, the corresponding MUS computation involved starting from SUSes containing only 24–31 constraints. In contrast, GPT-4 safely minimized 87.9% of the formulae for an average minimization of 73.4%; Gemini-1.5 Pro, 81.8% for an average minimization of 89%; and Claude-3.5 Sonnet, 87.9% for an average minimization of 98%. Thus, with Claude-3.5 Sonnet, the MUS computation for these formulae involved SUSes containing only 7–9 constraints, while the average size of these MUSes is 3.2. In terms of the search space for MUSes, *this represents a reduction from* $O(2^{50}) \rightarrow O(2^{31})$ *with GPT-3.5, and* $O(2^{50}) \rightarrow O(2^{9})$ *with Claude-3.5 Sonnet*.

Furthermore, for 244 string formulae, we noticed that Claude-3.5 Sonnet with CoT-$\mathcal{SE}$ minimizes *exactly* to their MUSes (*i.e.*, $m = 1$). In comparison, GPT-4, Gemini-1.5 Pro, and GPT-3.5 do so only 73, 57, and 9 times. Upon further inspection, we observed that *for* 80.9% *of the instances in the* 0–10 *constraint range, Claude-3.5 Sonnet safely minimizes exactly to the MUSes. Among the* 10–20*,* 20–30*,* 30–40*, and* 40–50 *constraint ranges, CoT-*$\mathcal{SE}$ *minimizes to the MUS via macro-reasoning for 69, 47, 72, and 18 instances, respectively.*

RA$_3$. SAFEMIN with Claude-3.5 Sonnet helps minimize infeasible string constraint systems to SUSes that are only ∼2% larger than the corresponding MUSes, reducing their search space from $O(2^{50}) \rightarrow O(2^{9})$ and localizing exactly to the MUSes in 62.9% of the cases.

VIII. DIVERSE REASONING PATHS FOR PARALLELIZED, PARTIAL ENUMERATION OF MULTIPLE MUSES

By design, SAFEMIN is able to *explore diverse reasoning paths to generate multiple SUS candidates*. Thus, for a decoding sample size of k in the Explorer LLM, it can generate $O(k)$ SUSes. In this experiment, we use Claude-3.5 Sonnet as the LLM within SAFEMIN and assess its ability to identify multiple SUSes in parallel for a given constraint set, which highlights its usefulness in producing non-unique MUSes.

A. Experiment Setup

Let us use $\mathcal{D}$ to denote our dataset of string formulae. The MUS computation from an input formula by using an SMT solver is deterministic, and exhibits a one-to-one correspondence. Accordingly, we compare the partial enumeration of MUSes by SAFEMIN along two dimensions, and establish loose lower and upper bounds for the total number of unique MUSes produced by the SMT solver as follows:

Our first experiment setting helps establish the benefits of using *self-exploration* in SAFEMIN toward partial enumeration. Specifically, for the lower bound, we extract MUSes for each of the input string formulae ($C \rightarrow \mathcal{M}$) from the SMT solver, as well as the candidate SUSes ($k = 5$) generated by SAFEMIN for each formula ($\forall_{i \leq k} \mathcal{S}_i \rightarrow \mathcal{M}$). Thus, it represents the comparison between the sets $\cup_j (C \rightarrow \mathcal{M}_j)$ and $\cup_j (\cup_i^k (\mathcal{S}_i \rightarrow \mathcal{M}_{ij}))$, where $j = 1..|\mathcal{D}|$.

The second setting acts as a benchmark for SAFEMIN. Specifically, we mimic the partial enumeration of the MUSes from the original constraints C by using k unique random seeds within SMT solver, and collect the MUSes ($\forall_{i \leq k} C_i \rightarrow \mathcal{M}_i$). The basis for selecting those k seeds is to make the comparison consistent with the k candidates SUSes, and establish a loose upper bound for all possible unique MUSes (having a search space of $O(2^{|C|})$). Thus, it represents the comparison between $\cup_j (\cup_i^k (C_i \rightarrow \mathcal{M}_{ij}))$ and $\cup_j (\cup_i^k (\mathcal{S}_i \rightarrow \mathcal{M}_{ij}))$ where j=1..$|\mathcal{D}|$.

B. Experiment Results

First, we conducted an overlapping analysis of MUS enumeration, using the SMT solver directly on the input formula (*i.e.*, $C \rightarrow \mathcal{M}$), and by combining GPT-4 in CoT-$\mathcal{SE}$ with the solver (*i.e.*, $\mathcal{S} \rightarrow \mathcal{M}$). Of the computed MUSes, we can see that 246 are common to both approaches. Furthermore, SAFEMIN helps localize **277** additional MUSes that were not captured by the solver directly, and misses 120 that were deterministically computed by the solver directly.

SAFEMIN also computes non-unique MUSes for 32.8% of the instances, resulting in it localizing 42.9% more MUSes than when the input formulae were solved directly. That is, in 32.8% of the instances, it explored different reasoning paths to successfully identify non-unique sources of inconsistencies.

In the second setting, with multiple runs of the SMT solver on the input formula, we observed that it computes a total of 887 distinct MUSes. Of these, **385** were also computed by SAFEMIN, and **138** were computed by SAFEMIN but not the solver. Overall, SAFEMIN computes 58.9% of the total MUSes in the benchmark. Thus, our findings corroborate our design that *self-exploration* helps the LLMs exploit diverse reasoning paths, facilitating a parallelized enumeration of MUSes.

RA$_4$. The SUSes from SAFEMIN helps SMT solver capture non-unique MUSes in 32.8% of the instances, thus computing 42.9% more MUSes than the original solver without SAFEMIN, and 58.9% of the total ones.

IX. Detection of Infeasible Paths in Source Code

```
1  public static byte opaques(byte x, byte y) {
2     byte z;
3     if ((byte) (151 * (39 * ((x ^ y) + 2 * (x & y))
4        + 23) + 111) >(byte) ((x ^ y) + 2 * (x & y)))
5     {
6        ...//BLOCK1
7     }
8     else if
9     ((byte) (x - y + 2 * (~ x & y) - (x ^ y)) == 0x17)
10    {
11       ...//BLOCK2
12    }
13    else if
14    ((byte) (195 + 97 * x + 159 * y + 194 *
15      ~ (x |~ y) + 159 * (x ^ y) + (163 + x + 255 * y +
16      2 * ~ (x |~ y) + 255 * (x ^ y)) * (232 + 248 * x +
17      8 * y + 240 * ~ (x |~ y) + 8 * (x ^ y)) - 57) < 100)
18    { ...//BLOCK3
19    } else
20    { ...//BLOCK4
21    }
22    return z;
23 }
```

Fig. 7. A case study on detecting infeasible paths in source code. Lines highlighted in *green* and *red* indicate feasible and infeasible paths, respectively.

In this section, we illustrate via a case study an application of SafeMin in *detecting infeasible paths in source code*. We selected a C code snippet from a website containing source code with infeasible paths [38]. We converted the code into Java (Fig. 7). The feasible paths are highlighted in green and infeasible paths in red. After converting the paths into the SMT-Lib format, we used CVC5 to test the path constraints.

The constraint system comprises three constraints, two of which, as in lines 3–4 and line 9, contain conflicts and are unsatisfiable, resulting in two MUSes. Here, we extracted all possible path constraints and input them to SafeMin. It successfully minimized the constraint system, and accurately identified the UNSAT-causing constraints, directly returning the MUSes and thereby *identifying the infeasible paths*.

This case study illustrates a useful scenario of SafeMin in identifying the infeasible path that could help developers in fixing the issue of unreachable code. It is able to safely minimize directly to a MUS, providing an explanation for developers in understanding the reason for a unfeasible path by pointing out the MUS with a minimum number of constraints. More in-depth experiments in this regard need to be conducted on more complex path constraints in large codebases.

X. Related Work

While not formally defined, the notion of *safe minimization* occurs in various software engineering tasks, including conditioned slicing [39], constraint slicing [40], symbolic execution [41], [42], [1], [2], [43]. However, they do not leverage any contextual information. There has been research along two dimensions to further optimize these processes.

The first dimension includes efforts to incorporate heuristic-based conflict analysis strategies in SMT solvers [8], [9], [10]. To this end, Lagniez *et al.* [12] proposed factoring out assumptions eagerly, while Belov *et al* [44] trimmed input formulae based on proofs. Marques-Silva *et al.* [45] proposed a tree search-based algorithm that constructs a solution through a combination of backtracking and learning new clauses from conflicts. These strategies are still limited and can not scale.

The next dimension includes the incorporation of *learning-based* components in CDCL solvers. This includes learning to predict conflict causing-variables to guide branch selection [14]. Wang *et al.* [13] focus on the bottlenecks in clause deletion, and predict what value a variable should have. The closest to our work is SatFormer [17], where Shi *et al.* try to directly predict the conflict-causing clauses. Nonetheless, these are trained on specific theories and can not generalize, thus limiting their extension to complex SMT theories.

In this work, we propose a fundamentally different approach, aiming to leverage the LLM's macro-reasoning capabilities. We are encouraged by recent research on automated reasoning with LLMs, where they have shown promising results [29], [29], [30], [33] and a potential to achieve scale.

XI. Threats to Validity

First, we only evaluated SafeMin on the LeetCode benchmarks in string theory. This choice is grounded in the inherent complexity of string constraints and the particular relevance of these formulae to software engineering, due to being sourced from real-world coding challenges. Besides, they involve both string operations (*e.g.*, concatenation, substring) and linear integer arithmetic operations (*e.g.*, length-based computations), highlighting SafeMin's potential applicability to other domains beyond string theory, the results for which might be different. Second, we used CVC5 [10] as the SMT solver in our experiments. Thus, the results for Section VII may vary with other SMT solvers. However, our "sample-and-enumerate" framework involving an LLM and SMT-based verifier is general and can be extended to all SMT solvers. Third, while we tested our approach on multiple LLMs, the results for others might vary and requires further investigation. Fourth, the manual investigation in Section VI might have human errors. Nonetheless, we employed multiple evaluators to reach a consensus. Finally, we showed SafeMin's usefulness in MUS computation and detecting infeasible paths. However, our approach to safe minimization could also motivate other tasks, including constraint-based program slicing, testing, *etc*.

XII. Conclusion

This study introduces SafeMin, which leverages LLMs to safely minimize formulae by employing macro-reasoning on constraint sub-groups. By exploring various reasoning paths through a "sample-and-enumerate" strategy, SafeMin generates multiple candidate subsets of the constraints while retaining conflict-causing ones. When applied to searching non-unique MUSes, it brings substantial advantages, reducing search space in 94.3% of instances by an average reduction of 98% and capturing 42.9% more MUSes than traditional solvers. We showed its usefulness in infeasible path detection.

References

[1] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[2] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012. [Online]. Available: https://doi.org/10.1145/2093548.2093564

[3] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111. Springer, 2005, pp. 364–387. [Online]. Available: https://doi.org/10.1007/11804192_17

[4] A. Griggio, "An effective SMT engine for formal verification," Ph.D. dissertation, University of Trento, Italy, 2009. [Online]. Available: http://eprints-phd.biblio.unitn.it/166/

[5] V. V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, 2008. [Online]. Available: https://doi.org/10.1109/TCAD.2008.923410

[6] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. S. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan, "Stratified abstraction of access control policies," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 12224. Springer, 2020, pp. 165–176. [Online]. Available: https://doi.org/10.1007/978-3-030-53288-8_9

[7] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic program optimization," *Commun. ACM*, vol. 59, no. 2, pp. 114–122, 2016. [Online]. Available: https://doi.org/10.1145/2863701

[8] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[9] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. W. Barrett, and M. Deters, "An efficient SMT solver for string constraints," *Formal Methods Syst. Des.*, vol. 48, no. 3, pp. 206–234, 2016. [Online]. Available: https://doi.org/10.1007/s10703-016-0247-6

[10] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *TACAS (1)*, ser. Lecture Notes in Computer Science, vol. 13243. Springer, 2022, pp. 415–442.

[11] W. Guo, H.-L. Zhen, X. Li, W. Luo, M. Yuan, Y. Jin, and J. Yan, "Machine learning methods in solving the boolean satisfiability problem," *Machine Intelligence Research*, vol. 20, no. 1, pp. 640–655, 2023.

[12] J. Lagniez and A. Biere, "Factoring out assumptions to speed up MUS extraction," in *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. Järvisalo and A. V. Gelder, Eds., vol. 7962. Springer, 2013, pp. 276–292. [Online]. Available: https://doi.org/10.1007/978-3-642-39071-5_21

[13] W. Wang, Y. Hu, M. Tiwari, S. Khurshid, K. L. McMillan, and R. Miikkulainen, "Neurocomb: Improving SAT solving with graph neural networks," *CoRR*, vol. abs/2110.14053, 2021. [Online]. Available: https://arxiv.org/abs/2110.14053

[14] D. Selsam and N. S. Bjørner, "Guiding high-performance SAT solvers with unsat-core predictions," in *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Janota and I. Lynce, Eds., vol. 11628. Springer, 2019, pp. 336–353. [Online]. Available: https://doi.org/10.1007/978-3-030-24258-9_24

[15] W. Zhang, Z. Sun, Q. Zhu, G. Li, S. Cai, Y. Xiong, and L. Zhang, "Nlocalsat: boosting local search with solution prediction," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, ser. IJCAI'20, 2021.

[16] V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro, "Can q-learning with graph networks learn a generalizable branching heuristic for a sat solver?" in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[17] Z. Shi, M. Li, Y. Liu, S. Khan, J. Huang, H. Zhen, M. Yuan, and Q. Xu, "Satformer: Transformer-based UNSAT core learning," in *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*. IEEE, 2023, pp. 1–4. [Online]. Available: https://doi.org/10.1109/ICCAD57390.2023.10323731

[18] M. Li, Z. Shi, Q. Lai, S. Khan, S. Cai, and Q. Xu, "Deepsat: An eda-driven learning framework for sat," 2023. [Online]. Available: https://arxiv.org/abs/2205.13745

[19] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=HJMC_iA5tm

[20] M. H. Liffiton and A. Malik, "Enumerating infeasibility: Finding multiple muses quickly," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, ser. Lecture Notes in Computer Science, C. P. Gomes and M. Sellmann, Eds., vol. 7874. Springer, 2013, pp. 160–175. [Online]. Available: https://doi.org/10.1007/978-3-642-38171-3_11

[21] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva, "Fast, flexible MUS enumeration," *Constraints An Int. J.*, vol. 21, no. 2, pp. 223–250, 2016. [Online]. Available: https://doi.org/10.1007/s10601-015-9183-0

[22] J. Bendík, I. Cerná, and N. Benes, "Recursive online enumeration of all minimal unsatisfiable subsets," in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 11138. Springer, 2018, pp. 143–159. [Online]. Available: https://doi.org/10.1007/978-3-030-01090-4_9

[23] W. Zhao and M. H. Liffiton, "Parallelizing partial MUS enumeration," in *ICTAI*. IEEE Computer Society, 2016, pp. 464–471.

[24] J. Bendík and K. S. Meel, "Approximate counting of minimal unsatisfiable subsets," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 12224. Springer, 2020, pp. 439–462. [Online]. Available: https://doi.org/10.1007/978-3-030-53288-8_21

[25] P. Lymperopoulos and L. Liu, "Graph pruning for enumeration of minimal unsatisfiable subsets," *arXiv preprint arXiv:2402.15524*, 2024.

[26] (2024) Replication package for intellismt. [Online]. Available: https://github.com/se-doubleblind/intellismt

[27] (2024) Leet code. [Online]. Available: https://leetcode.com

[28] (2024) Smt-lib benchmarks. [Online]. Available: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA

[29] A. Lewkowycz, A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo, Y. Wu, B. Neyshabur, G. Gur-Ari, and V. Misra, "Solving quantitative reasoning problems with language models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022.

[30] T. Morishita, G. Morio, A. Yamaguchi, and Y. Sogawa, "Learning deductive reasoning from synthetic corpus based on formal logic," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, ser. Proceedings of Machine Learning Research, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202. PMLR, 2023, pp. 25 254–25 274. [Online]. Available: https://proceedings.mlr.press/v202/morishita23a.html

[31] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. Canton-Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," *CoRR*, vol. abs/2307.09288, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.09288

[32] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. M. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with GPT-4," *CoRR*, vol. abs/2303.12712, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.12712

[33] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," 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. 1229–1241. [Online]. Available: https://doi.org/10.1145/3611643.3616243

[34] M. H. Liffiton and A. Malik, "Enumerating infeasibility: Finding multiple muses quickly," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, C. Gomes and M. Sellmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–175.

[35] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2022.

[36] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022.

[37] (2023) String benchmark set smt-comp. [Online]. Available: https://smt-comp.github.io/2023/

[38] A. G. i Montolio. (2023) A gentle introduction to smt-based program analysis. Accessed: 2024-07-25. [Online]. Available: https://furalabs.com/blog/2023/02/12/intro_to_smt_analysis

[39] G. Canfora, A. Cimitile, and A. D. Lucia, "Conditioned program slicing," *Inf. Softw. Technol.*, vol. 40, no. 11-12, pp. 595–607, 1998. [Online]. Available: https://doi.org/10.1016/S0950-5849(98)00086-X

[40] A. D. Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," in *4th International Workshop on Program Comprehension (WPC '96), March 29-31, 1996, Berlin, Germany*. IEEE Computer Society, 1996, pp. 9–10. [Online]. Available: https://doi.org/10.1109/WPC.1996.501116

[41] R. Aïssat, F. Voisin, and B. Wolff, "Infeasible paths elimination by symbolic execution techniques: Proof of correctness and preservation of paths," *Arch. Formal Proofs*, vol. 2016, 2016. [Online]. Available: https://www.isa-afp.org/entries/InfPathElimination.shtml

[42] I. Erete and A. Orso, "Optimizing constraint solving to better support symbolic execution," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. IEEE Computer Society, 2011, pp. 310–315. [Online]. Available: https://doi.org/10.1109/ICSTW.2011.98

[43] E. Bounimova, P. Godefroid, and D. A. Molnar, "Billions and billions of constraints: whitebox fuzz testing in production," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 122–131. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606558

[44] A. Belov, M. Heule, and J. Marques-Silva, "MUS extraction using clausal proofs," in *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, ser. Lecture Notes in Computer Science, C. Sinz and U. Egly, Eds., vol. 8561. Springer, 2014, pp. 48–57. [Online]. Available: https://doi.org/10.1007/978-3-319-09284-3_5

[45] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 133–182. [Online]. Available: https://doi.org/10.3233/FAIA200987