



# Exploring Large Language Models for Requirements on String Values

Aren A. Babikian\*   
Dep't of Computer Science  
University of Toronto  
Toronto, Canada  
babikian@cs.toronto.edu

Boqi Chen\*   
Electrical and Computer Engineering  
McGill University  
Montreal, Canada  
boqi.chen@mail.mcgill.ca

Gunter Mussbacher   
Electrical and Computer Engineering  
McGill University  
Montreal, Canada  
gunter.mussbacher@mcgill.ca

**Abstract**—Behavior-driven development (BDD) enables collaboration among different stakeholders by employing a natural-language representation of system requirements and of test scenarios. These scenarios often involve constraints over string values, e.g. for the validity of email addresses, which are challenging to test comprehensively. Traditional methods like SMT solvers (e.g. Z3, Ostrich) handle constraints efficiently but produce unrealistic strings and require formal specifications that are often unavailable and expensive to compute. This paper explores the potential of large language models (LLMs) in generating realistic, constraint-satisfying strings for BDD. We propose an evaluation framework to assess LLMs' ability to (1) generate consistent string values and (2) detect constraint inconsistencies. In our experiments, three LLMs are compared to state-of-the-art solvers using constraints from a software engineering course project. Results show that while solvers dominate in precision and recall, LLMs derive realistic strings more suitable for a requirements engineering context. With these trade-offs, we believe that, when formal constraints are available, a combined LLM-solver approach could offer a more effective solution.

**Index Terms**—large language model, specification-based testing, constraint solving

## I. INTRODUCTION

**Motivation.** With the rise of agile methodologies, effective software development relies on the adequate interaction between domain experts, technical team members and end-users. This interaction becomes particularly challenging when considering the varying levels of domain expertise and technical knowledge of involved parties. To facilitate this interaction, agile methodologies often integrate behavior driven development (BDD) approaches, where requirements of the system behavior are represented at a high level of abstraction (often in natural language) understandable by all parties [1].

As part of the BDD workflow, system requirements yield natural-language test scenarios which express the expected behavior of the system-under-test in a semi-structured format. The Gherkin language [2] is often used to represent such test scenarios using a *Given-When-Then* structure: *Given* some initial context; *When* some event occurs; *Then* this should result in some expected outcome.

As an example, let us consider a software system where users may create an email address. To test such

a system, a tester may create the following test scenario: *Given* a user is on the registration page *And* the user has entered "dummy@email.com" in the email field *And* "dummy@email.com" is not registered in the system; *When* the user clicks on the "create an email address" button; *Then* the system should successfully create the email address "dummy@email.com".

**Problem statement.** While representing tests in natural language does simplify the creation of individual test cases, creating test suites to adequately evaluate a software system remains a daunting and error-prone task. Internal logic of software systems is often complex, thus adequately testing the system requires a set of specific inputs (e.g. with respect to some requirements). For instance, the system should not allow users to create email addresses from entries such as "dummyemail.com" or "dummy@emailcom": email addresses are required to contain exactly one "@" character and at least one "." character following it.

To adequately handle natural-language test cases, BDD relies on a string representation. As a result, adequate testing within BDD requires input string data (in our case, the email entry) that satisfies some constraints (e.g. derived from system requirements). While generated strings must satisfy all input constraints, this may not be possible in cases where requirements are inconsistent. Such inconsistencies must be identified early, as addressing them in later stage can be expensive.

To generate constraint-satisfying strings, existing approaches are often formal: they rely on general-purpose SMT solvers such as Z3 [3] or string-specific constraint solvers, such as Ostrich [4]. Such approaches can handle complex string constraints efficiently, but their derived strings lack realism, which is particularly problematic for natural language test case creation. Moreover, such approaches rely on a formal definition of requirements (e.g. using the SMT-LIB2 syntax), which is often unavailable (and hard to create) for many software projects. Language models (LMs) are primarily designed to generate natural, contextually relevant text. However, adapting them for string constraint satisfaction typically requires extensive training or fine-tuning; a process that demands large datasets, which are often unavailable. Recently, large language models (LLMs) have emerged as a promising alternative for logic reasoning tasks [5], particularly due to their strong

\*Both authors contributed equally to the research.

few-shot generalization capabilities in data-scarce settings. However, their effectiveness in satisfying strict specification constraints remains less studied.

**Contributions.** This paper explores the ability of LLMs (vis-à-vis formal approaches) to generate string values that satisfy a set of input constraints and to detect inconsistencies in string constraint sets. The specific contributions of this paper are the following.

- We propose a novel approach to evaluate whether LLMs are capable of (1) generating consistent string values and (2) detecting inconsistencies for natural-language string constraint satisfaction problems given as input.
- Our approach integrates a constraint solver (1) for validating the LLM output, and (2) for string constraint solving independently when formal specifications exist.
- We conduct experiments with three popular LLMs, (i.e. GPT-4o, GPT-4o-mini, Llama3.1-8b [6]) and compare their performance to that of three state-of-the-art formal approaches (i.e. Z3 [3], Z3str [7], cvc5 [8]). We assess a case study consisting of constraints over six types of string variables drawn from an undergraduate software engineering course project.

**Added value.** To our best knowledge, our paper is the first to evaluate the capacity of LLMs for solving formal constraint satisfaction problems over strings. Through our comparative evaluation, we provide insights on the strengths and weaknesses of both LLMs and formal approaches in generating consistent strings in the context of BDD. Additionally, we provide insights on the usability of generated strings for requirements specification, e.g., by assessing their realism.

## II. BACKGROUND

### A. Behavior-driven development (BDD)

To illustrate the workflow employed by the BDD methodology, we expand on the example depicted in Section I. First, let us consider that the email address creation system has the following three requirements:

- R1 The system shall require an email address to contain exactly one "@" character.
- R2 The system shall require an email address to not end with a "." character.
- R3 The system shall require an email to contain at least one "." character after the "@" character.

Given these requirements, a developer must first select a set of test scenarios and associate an expected system behavior. An example containing four such scenarios, represented as conditions over the requirements, is shown in Table I. Once a scenario is selected, the developer must define an input string that would trigger the adequate portion of the underlying system. With an increased number of requirements, not only does the number of relevant scenarios increase but so does the complexity of the string constraint satisfaction problem.

The problem becomes particularly challenging when considering the existence of string-specific corner cases as part of the test suite. For instance, it is difficult to manually

understand that a  $R1 \wedge \neg R2 \wedge \neg R3$  scenario is unsatisfiable, since a string cannot end with a "." ( $\neg R2$ ) if it does not contain any "." characters after "@" ( $\neg R3$ ). To address this challenge, we leverage automated approaches (LLMs, SMT solvers) for generation of consistent strings as well as for automated detection of (unsatisfiable) corner cases.

TABLE I  
TEST SCENARIOS AND CORRESPONDING STRING INPUTS DERIVED FOR

Scenarios	System Behavior	Input
$R1 \wedge R2 \wedge R3$	Store email in database	"dummy@email.com"
$\neg R1 \wedge R2 \wedge R3$	Show error message	"dummy@@email.com"
$R1 \wedge \neg R2 \wedge R3$	Show error message	"dummy@email."
$R1 \wedge R2 \wedge \neg R3$	Show error message	"dummy@emailcom"

### B. Constraint satisfaction problems over strings

String-based constraints are common not only in commercial software applications, e.g., through password and email restrictions, but are also addressed significantly in scientific research. Automated string constraint solvers [4], [9]–[12] often rely on formal reasoning techniques and are designed to solve collections of complex constraints. These approaches are evaluated over large benchmarks<sup>1</sup> and in the context of solver competitions<sup>2</sup>, where string constraints are represented using a formal language (e.g. SMT-LIB2 [13]).

In the context of these complex benchmarks, solvers are assessed in accordance to their achieved runtime and success rate. Non-functional parameters, such as qualitative analysis (e.g. realism, relevance) of the generated string, are not considered, which limits the extensibility of formal solvers to domains such as requirement engineering. Additionally, solvers require a formal problem definition as input, which is rarely available for systems employing the BDD framework.

### C. Large language models (LLMs)

LLMs are advanced neural networks based on the transformer architecture [14], initially designed for language modeling tasks. Through instruction fine-tuning, LLMs can be trained to follow input instructions, enabling them to adapt to diverse tasks via *in-context learning* using a few examples or even zero-shot predictions with only task descriptions.

Recent research highlights LLM’s emerging reasoning capabilities, achieving notable performance on many logic reasoning benchmarks [5]. This positions them as promising alternatives to traditional constraint solvers for addressing constraint-solving problems involving string values.

Since LLMs accept natural language as input, a significant amount of research has been conducted to optimize prompts for improving performance, an active area known as *prompt engineering*. Techniques such as zero-shot chain-of-thought reasoning [15] and leveraging emotional stimuli [16] have proven effective across various LLMs.

Although LLMs typically produce natural language output, recent advancements have enabled them to produce structured

<sup>1</sup><https://smt-lib.org/benchmarks.shtml>

<sup>2</sup><https://smt-comp.github.io/2024/>

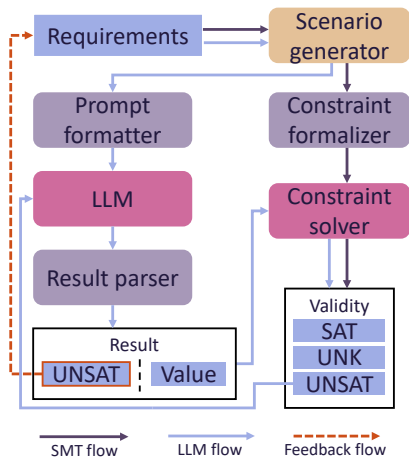


Fig. 1. Overview of the approaches

output, such as the *structured outputs* of OpenAI API<sup>3</sup> and *output parsers* from LangChain<sup>4</sup>. In this paper, we use LangChain to instruct the LLM to present final answers in JSON format.

### III. PROPOSED ARCHITECTURE

Figure 1 illustrates the process of test data generation using both LLMs and constraint solvers. The purple arrows indicate the flow for constraint solvers while the blue arrows represent the flow for LLMs. The process begins with **scenario generation** based on the requirements. These generated scenarios are then provided to both the constraint solver and to the LLM for test data generation. The test data produced by the LLM is validated using the constraint solver. If necessary, the LLM is prompted to retry. If the LLM is unable to produce a valid value, this may suggest potential inconsistencies in the requirements (as indicated by the red dashed arrow).

#### A. Scenario generation

We focus on natural language requirements for strings. Given a requirement, the *scenario generator* formulates it as a list of conditions in a behavior-testing scenario that the string variable must satisfy. The goal of the generator is to either produce test data that meets all constraints and can be used in the scenario or flag the scenario as unsolvable (UNSAT).

#### B. Data generation with constraint solvers

To generate string data using a constraint solver, the conditions are first translated into a formal specification through a *constraint formalizer*. Since mapping natural language to constraint languages is not the focus of this paper, we manually translate the conditions into *SMT-LIB2* constraints.

For each set of constraints associated with a scenario, a constraint solver produces one of three possible outputs: (1) **SAT**, indicating that the constraints are satisfiable with a **valid string value**; (2) **UNSAT**, indicating that the constraints are inconsistent and unsatisfiable; or (3) **UNK**, indicating that the solver fails to make a decision within a pre-defined time limit.

<sup>3</sup><https://openai.com/index/introducing-structured-outputs-in-the-api/>

<sup>4</sup><https://python.langchain.com/>

```

You are a test engineer working on creating test data for
a new feature. You are given a variable "{name}" with some
associated constraints.

First, explain the meaning of each constraint. Then think
step by step to find a string value for "{name}" that
satisfy ALL following constraints:
{constraints}
If the word "{name}" is meaningful, the value should be as
realistic for "{name}" as possible.

The output should follow the following format. If no value
can satisfy all constraints, assign the value "UNSAT":
{output_format}

Keep the results concise. If the answer is not correct,
then you will be fired from your job.

```

Fig. 2. Prompt template used for LLMs

#### C. Data generation with LLMs

LLMs can directly generate string values based on text conditions in scenarios. The *prompt formatter* constructs a prompt for LLMs by using the input scenario and the string variable's name to guide the generation of realistic values. Figure 2 illustrates the prompt template used for generation.

We adopt *zero-shot* prompting, focusing on identifying a single valid string value for each scenario. To enhance the LLM's performance, we incorporate several best practices into the prompt design. The prompt begins with a *role description*, establishing the task context. To improve reasoning, we enable the *zero-shot chain-of-thought* [15] by instructing the LLM to think step by step. Additionally, *emotional stimuli*, shown to improve LLM performance across various tasks [16], are appended at the prompt's end. The scenario generator provides the string variable's *name* and *constraints*.

Since LLMs generate text-based responses, a *result parser* is used to extract the values produced by the model. To facilitate straightforward extraction, the LLM is instructed to output responses in a JSON format specified by the *output format*. If the LLM fails to generate a valid JSON, it is prompted to retry until a valid response is produced.

The LLM's output can fall into two types: (1) a string value that *may* satisfy the input constraints or (2) **UNSAT** indicating that the scenario *may* be unsolvable. The correctness of LLM outputs cannot be guaranteed, hence we use a constraint solver to validate them. When the LLM generates a value for the string variable, a new constraint is introduced in the constraint solver input requiring the string variable to equal the LLM-generated value. The value is considered valid if the constraint solver returns a **SAT** result, otherwise it is invalidated. This mechanism can also act as a filtering step, prompting the LLM to retry if the generated value is invalid.

#### D. Consistency feedback

When the constraint solver returns an UNSAT result for a scenario, it indicates the presence of inconsistent constraints within the scenario. Similarly, as the LLM operates based on the natural language specifications, if the LLM generates an UNSAT value, this may reflect inconsistencies in the original *requirements*. This capability is particularly valuable in cases where natural language requirements are challenging to for-

malize as constraints. In this paper, we target to investigate LLMs’ ability to perform such consistency checks.

#### IV. DATASET

##### A. Data collection

To evaluate the performance of string value generation approaches, we collect six different types of string variables. Requirements for these variables are extracted from past projects from an undergraduate software engineering course. Table II shows these types of string variables and the number of requirements associated with each type. From these requirements, we derive a total of 192 test scenarios (string constraint satisfaction problems) by considering all possible combinations of the requirements and their negations using *equivalence partitioning* [17]. To ensure conditions are precise, we manually formulate these requirements. While some problems may be unsatisfiable, we integrate them as a manner of examining the capacity of LLMs to detect such inconsistencies in requirements.

##### B. Ground truth labeling

Before using LLMs, we examine the performance of constraint solvers in producing the ground truth labels.

**Constraint solvers:** We manually translate the requirements to formal logical constraints and use **Z3** [3] to derive consistent string values from them. However, **Z3** is a general-purpose SMT-solver and not specialized to handle constraints related to string variables. Thus, in addition, we also evaluate two solvers specialized for strings: **Z3str** [7] and **cvc5** [8]. We set a timeout of 30 seconds for each run of the constraint solvers.

**Results:** Table III shows the labeling results of each solver. In general, the **cvc5** solver significantly outperforms the other two solvers with only 3 timeout cases: it is particularly better at identifying unsolvable cases. Interestingly, all 3 unknown cases are associated with constraints on *date* strings, highlighting handling dates with the constraint solver can be non-trivial.

Analysis of our results shows that one UNK case from **cvc5** is solved by **Z3**. Furthermore, upon manual inspection, we determine that the remaining two UNK cases are in fact UNSAT (and we label them accordingly). As such, the dataset contains in total 140 SAT problems and 52 UNSAT problems. This shows that UNSAT problems are common for string-based constraints in the context of BDD, thus making their early detection a relevant challenge.

#### V. EVALUATION

In this section, we present our evaluation results pertaining to the use of LLMs for solving string constraint satisfaction problems derived from requirements. We introduce our evaluation setup, built upon the dataset described in Section IV, we present our measurement data and we discuss the implications of our findings in the context of requirements engineering. Specifically, we address the following research questions:

RQ1 How do LLMs compare with constraint solvers in (RQ1.1) generating consistent strings and (RQ1.2) detecting inconsistent specifications?

RQ2 How does the success rate of LLMs improve with increased attempts?

##### A. Evaluation setup

*a) Compared methods:* As part of our evaluation, we evaluate three popular LLMs and compare their performance to those of the three constraint solvers mentioned in Section IV. Among the evaluated LLMs, **GPT4o** and **GPT4o-mini**<sup>5</sup> are the latest powerful large language models developed by OpenAI. They regularly achieve improved performance in multiple tasks compared with other LLMs, especially in logical reasoning-related tasks, making them most suitable for this study. However, powerful LLMs have some practical limitations. They can be either (1) very expensive to run or (2) operated behind an API with potential privacy concerns. With these limitations in mind, we also consider a less-powerful, yet low-cost, open-source LLM **Llama3.1-8b** [6] with 4-bit quantization, runnable in most consumer-grade GPUs.

To validate the LLM-generated strings, we integrate the **Z3** solver. Despite resulting in numerous UNK outcomes when attempting to solve the constraint satisfaction problems, **Z3** always terminates (with a SAT or UNSAT) outcome when used for validation purposes.

*b) Metrics:* We use *generation success rate (GSR)* to evaluate the performance of valid string generation. The success rate evaluates the percentage of cases where the approach generates a string satisfying all constraints. The fact that a constraint-solving problem is UNSAT may indicate that there are some inconsistencies in the requirements. We evaluate the performance of LLMs in identifying the UNSAT cases with recall (**UNSAT-R**) and precision (**UNSAT-P**).

*c) Implementation details:* For LLMs, we use a temperature value of 0.7 for **GPT** models and 0.8 for **Llama3.1-8b**. The detailed implementation and the experiment setup can be found in our GitHub repository<sup>6</sup>.

##### B. RQ1: Comparative analysis

RQ1 aims to evaluate the ability of LLMs, when given a set of string constraints as input, (RQ1.1) to generate a consistent string when possible or (RQ1.2) to detect any inconsistencies in the input constraints. The performance of LLMs is compared to that of state-of-the-art SMT solvers. Comparative results, alongside standard deviation values for LLMs (over 10 runs) are presented in Table IV.

*a) RQ1.1 results:* When attempting to solve consistent string constraint satisfaction problems, all three SMT solvers provide high success rates (over 94%). **cvc5** achieves the highest success rate by solving all but one of the 140 consistent cases. In contrast, LLMs provide varying performance level when addressing consistent problems. As expected, larger models provide better performance than smaller models, with GSR ranging from 40.56% for **Llama3.1-8b** to 79.30% for **GPT4o**. In all cases, SMT solvers dominate LLMs in generating consistent strings.

<sup>5</sup><https://openai.com/index/hello-gpt-4o/>

<sup>6</sup><https://github.com/20001LastOrder/llm-string-constraints>

TABLE II  
DATASET STATISTICS AND GENERATED EXAMPLES

String Type	# of req.s	Example requirement (i.e., constraint)	LLM results (GPT-4o)	Solver results (cvc5)
Email	6	The system shall require an email address to contain exactly one "@" character.	john.doe@example.com	B@.A
Name	6	... a name to contain at least one space.	John Doe	Z Z
Password	5	... a password to contain one of the following characters: "!", "#", "\$".	Ab1\$	Z9a!
Url	3	.. a url to start with "http://" or "https://".	https://example.com	http://.
Date	4	... a date to contain two hyphens (-).	2023-10-15	0-1-1
International Bank Account Number	3	... the length of a bank number to be 22.	1210000000000000000000	121000AAAAAAAAAAAAAAAAAAAA

TABLE III  
GROUND TRUTH LABELING FOR EACH CASE IN THE DATASET

Solver	SAT	UNSAT	UNK
<b>Z3</b>	134	28	30
<b>Z3str</b>	132	29	31
<b>cvc5</b>	139	50	3
Final labeling	140	52	-

TABLE IV  
PERFORMANCE OF COMPARED APPROACHES (IN %)

Approach	GSR	UNSAT-R	UNSAT-P
<b>Z3</b>	95.71	53.85	100
<b>Z3str</b>	94.29	55.77	100
<b>cvc5</b>	99.29	96.15	100
<b>GPT4o</b>	79.30 ± 1.31	65.80 ± 0.60	69.74 ± 1.61
<b>GPT4o-mini</b>	57.54 ± 3.36	82.60 ± 2.01	48.49 ± 2.55
<b>Llama3.1-8b</b>	40.56 ± 4.23	24.00 ± 4.47	63.14 ± 7.77

**Answer to RQ1.1.** When addressing string constraint satisfaction problems, SMT solvers provide better performance (over 94% GSR) than LLMs (between 40% and 80% GSR). Additionally, we notice that larger models provide better GSR for consistent string generation.

*b) RQ1.2 results - SMT:* As opposed to results for RQ1.1, inconsistency detection yields varying performance from SMT solvers. **Z3** and **Z3str** detect inconsistent specifications with 53.85%-55.77% success rate, while **cvc5** detects 96.15% of inconsistencies. In cases where SMT solvers are unable to detect an inconsistency, they return a UNK outcome. Naturally, SMT solvers never return UNSAT for satisfiable specifications (UNSAT-P is 100% for all solvers).

Upon manual analysis, we determine that **Z3**-based solvers fail to identify inconsistencies in 8-9 *name* specifications, 18-20 *email* specification and 3 *date* specifications. As a key distinction, these string types include *relational* constraints over characters within the string. For instance, the *email* type contains a requirement stating that *The string must contain at least one "." character after the "@" character*. This requires the solver to (1) find the "@" character, (2) find the substring defining the part *after* "@", and (3) determining if the substring contains a ".". As indicated by our result, **Z3**-based solvers struggle with such complex requirements.

*c) RQ1.2 results - LLM:* LLMs provide varying performance when dealing with inconsistencies. GPT-based LLMs provide an improved recall value compared to **Z3**-based approaches, with an increase of 10.03% to 28.75%. **GPT4o-mini** provides the highest recall among LLMs (82.60 %), while **Llama3.1-8b** provides a recall of only 24.00%. In all cases, recall for LLMs are dominated by that of **cvc5**.

When assessing precision of UNSAT outcomes, **GPT4o** provides the best performance (69.74%), while the results for **Llama3.1-8b** (63.14%) are lower but comparable. Despite the high recall, **GPT4o-mini** achieves the worst precision (48.49%). We also note that **Llama3.1-8b** results show a larger standard deviation (7.77%) compared to all other measurements. We attribute this to the small, quantized size of **Llama3.1-8b**, which may tend to output "UNSAT" mentioned in the prompt rather than solving the problem. In all cases, precision results are dominated by SMT solvers providing formal soundness guarantees by construction.

**Answer to RQ1.2.** On one hand, while **GPT4o** and **GPT4o-mini** perform 10.03% to 28.75% better than **Z3**-based solvers in correctly detecting real UNSAT cases, recall-based measurements are dominated by **cvc5**. On the other hand, while LLMs provide precision between 48.49% and 69.74%, they are dominated by SMT solvers, which provide formal soundness guarantees by construction. As a practical outcome, we determine that **GPT4o**-based solvers are better suited among LLMs for inconsistency detection in constraint specifications.

### C. RQ2: Impact of attempts

When formal constraints are derived from requirements, constraint solvers can act as a *filtering validator* to reject invalid values. In this case, one can continuously ask the LLM to attempt to generate a value until it satisfies all constraints. Thus, RQ2 aims to evaluate the impact of such continued LLM attempts at generating string values with constraints as input.

To evaluate the effects of a sequence of generation attempts on success rate, we measure the **GSR@k** metric (used in similar settings [18].) which considers a case to be successful if one of **k** attempts is successful.

*a) Results:* Figure 3 shows the trend of **GSR@k** up to 10 attempts. As expected, the success rate increases as the number of attempts increases for all LLMs. In general, the successful

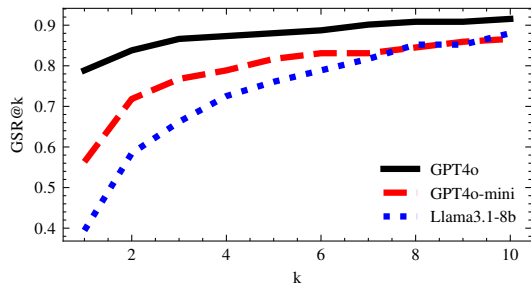


Fig. 3. Generation success rate after number of attempts for different LLMs

rate is increased for **GPT4o** by 12.68%, for GPT-4o-mini by 30.28%, and Llama3.1-8b by 48.59%.

Surprisingly, the improvement for **Llama3.1-8b** is much more significant compared with more powerful models. The value of GSR@10 for **Llama3.1-8b** is much closer to the GPT models and even beats **GPT-4o-mini** while only achieving about half of the **GPT-4o-mini** performance with one attempt. This observation suggests that **Llama3.1-8b** is in general less certain about its generated solutions. We observe a great variance in solution success rate over the sequence of attempts

Nevertheless, the improvement in performance for all LLMs starts to plateau around 8 attempts. Specifically, there are 8 common cases that *all* LLMs still fail to generate after 10 attempts, suggesting other types of approaches are necessary for further improvement. Compared to solver performance in Table IV, the best success rate after 10 attempts (91.55%) is still less than the worst constraint solver (with one attempt).

**Answer to RQ2.** The success rate for all LLMs increases as the number of attempts increases. Notably, the performance change in weaker LLMs (e.g., **Llama3.1-8b** by 48.59%) is much larger compared to more powerful LLMs (e.g. **GPT-4o** by 12.68%). These changes become less significant and start to plateau after 8 generations.

#### D. Discussion

**Assessing realism:** While our reported results indicate that SMT solvers are superior to LLMs in string consistency, manual analysis shows that strings from LLMs are often more realistic than those derived from SMT solvers. Some example values generated by both approaches are shown in Table II. Including additional constraints may help SMT solvers generate more realistic strings, but this would greatly complicate the constraint satisfaction problem. Moreover, this would require one to formalize the concept of realism, which is challenging due to its qualitative and subjective nature, particularly when attempting to minimize bias. As LLMs have been trained on natural language data, they can generate realistic strings when instructed. As such, when LLMs successfully generate consistent strings, they are better suited for applications in natural-language domains, such as requirements engineering.

During early stage of the research, we find that removing string variable name from the requirement generally reduces

the realism of the generated values, suggesting requirement quality may also influence the LLM’s performance.

**Detecting inconsistencies:** For inconsistent specifications, both recall and precision are dominated by **cvc5**. since no valid string may be generated for inconsistent specifications, qualitative analysis does not apply. As such, in cases where a formalization of the string constraint satisfaction problem exists, it is preferable to use SMT solvers, namely **cvc5**, for inconsistency detection, particularly due to their sound nature.

**A hybrid approach:** Considering these trade-offs, we deduce that, *when string constraint formalizations exist*, a *hybrid approach* would be most beneficial to address the challenge of string constraint solving in a requirements engineering context. Some early benefit is demonstrated by RQ2 when applying the solvers as a filtering validator for the LLM. Such an approach would first leverage an SMT solver to (quickly and accurately) determine whether a problem is satisfiable, and then employ some iterated approach like our proposed LLM-and-validation approach to derive realistic, consistent strings. Additionally, considering the benefits of constraint formalization, we may also investigate the capacity of LLMs to derive such formal specifications independently.

#### E. Threats to validity

**Internal validity:** The authors manually transform natural-language requirements into equivalent SMT-LIB2 constraints, which may introduce bias. To mitigate this bias and ensure correct transformation, we manually analyze the SMT-generated strings and UNSAT outcomes to ensure their correspondence to the natural-language requirements. In the case of LLMs, their output may slightly vary between each run. To address this variation, we evaluate the performance of LLMs over multiple runs and report the mean value and standard deviation. In some cases, such variation is even desirable (e.g. for GSR@k).

**External validity:** String values collected for experiments are from a software engineering course, i.e., representative for string requirements in education scenarios. The findings might slightly differ for other types of string requirements but we do not expect a significant difference.

**Construct validity:** We evaluate the performance of approaches on standard success rate for generation, and precision/recall for classification. When a validator exists, we use GSR@k, a metric widely used in similar settings.

## VI. RELATED WORK

**LLM for requirements engineering.** LLMs are becoming increasingly popular in various stages of requirements engineering, including elicitation, modeling and verification [19].

In requirement elicitation, LLMs assist in proposing interview questions [20] for identifying new requirements. Ronanki et. al. [21] explore the strengths and limitations of LLMs in the elicitation process. In requirement modeling, studies have shown LLM’s ability in generating goal models [22], [23]. Since requirements often contain ambiguities or inconsistencies that are difficult to detect manually, LLMs have been effectively applied to help identify these issues [24], [25].

In this paper, we explore the capability of LLMs to generate test data based on constraints derived from requirements. We also analyze how LLMs can provide insights into potential inconsistencies by marking certain cases as unsolvable.

**LLM for test data generation.** Due to their impressive natural language understanding and generation capability, LLMs have been used to generate test data from requirements for various software testing tasks, such as testing graphic user interfaces [26], fuzz testing [27] and even validating SMT solvers [28]. However, less focus has been on generating test cases for requirements on string variables.

Liu et al. [29] propose a method to generate test cases for string variables from user-written constraints. However, their approach focuses on natural language requirements that are less strict. In contrast, this paper also explores generating test cases with formal specifications, such that constraint solvers can be used to validate the results produced by LLMs.

## VII. CONCLUSION

This paper investigate LLMs for supporting requirement engineering in BDD testing scenarios. We evaluate LLMs' ability to (1) generate string values that satisfy constraints derived from requirements and to (2) identify inconsistencies within those requirements. Compared to traditional solvers, LLMs produce more realistic strings but struggle to fully satisfy specified constraints and to identify inconsistencies. In cases where formal constraints exists, integrating a solver as a filtering validator notably improves the success rate of LLMs after multiple attempts. These findings suggest that a hybrid approach, combining LLMs and solvers, could offer a more effective solution for consistent string generation in the context of requirements engineering.

As part of our future work, we plan to extend the approaches to handle more complex, interdependent constraints (e.g. if user A is the parent of user B, then the birth date of user A must be *before* that of user B). Furthermore, we plan to leverage LLMs to derive SMT-solver-compliant formulae from natural language requirements in order to automate the end-to-end string data generation process.

## REFERENCES

- [1] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Characterising the Quality of Behaviour Driven Development Specifications," in *Agile Processes in Software Engineering and Extreme Programming*, V. Stray, R. Hoda, M. Paasivaara, and P. Kruchten, Eds. Cham: Springer International Publishing, 2020, pp. 87–102.
- [2] M. Wynne and A. Hellesøy, *The Cucumber Book: Behaviour-driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.
- [3] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS 2008*. Springer, 2008, pp. 337–340.
- [4] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, "Decision procedures for path feasibility of string-manipulating programs with complex operations," *OSTRICH String Constraint Solver and Results*, vol. 3, no. POPL, pp. 49:1–49:30, Jan. 2019.
- [5] J. Huang and K. C.-C. Chang, "Towards reasoning in large language models: A survey," in *ACL 2023*. ACL, 2023, pp. 1049–1065.
- [6] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan et al., "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.
- [7] M. Berzish, V. Ganesh, and Y. Zheng, "Z3str3: A string solver with theory-aware heuristics," in *FMCAD 2017*. IEEE, 2017, pp. 55–59.
- [8] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli et al., "cvc5: A versatile and industrial-strength smt solver," in *TACAS 2022*. Springer, 2022, pp. 415–442.
- [9] Y.-F. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síč, "Solving string constraints with lengths by stabilization," *Proceedings of ACM Programming Languages*, vol. 7, no. OOPSLA2, Oct. 2023.
- [10] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka, "Solving String Constraints Using SAT," in *CAV 2023*. Springer, 2023, pp. 187–208.
- [11] H. Wu, Y.-F. Chen, Z. Wu, B. Xia, and N. Zhan, "A decision procedure for string constraints with string/integer conversion and flat regular constraints," *Acta Informatica*, vol. 61, no. 1, pp. 23–52, Mar. 2024.
- [12] S. Krings, J. Schmidt, P. Skowronek, J. Dunkelau, and D. Ehmke, "Towards Constraint Logic Programming over Strings for Test Data Generation," in *Declarative Programming and Knowledge Management*. Springer, 2020, pp. 139–159.
- [13] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017.
- [14] A. Vaswani, N. M. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.
- [15] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *NeurIPS 2022*, pp. 22 199–22 213, 2022.
- [16] C. Li, J. Wang, Y. Zhang, K. Zhu, W. Hou, J. Lian, F. Luo, Q. Yang, and X. Xie, "Large language models understand and can be enhanced by emotional stimuli," *arXiv preprint arXiv:2307.11760*, 2023.
- [17] I. Burnstein, *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.
- [18] M. Chen, J. Tworek, H. Jun, Q. Yuan et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
- [19] C. Arora, J. Grundy, and M. Abdelrazek, "Advancing requirements engineering through generative ai: Assessing the role of llms," in *Generative AI for Effective Software Development*. Springer, 2024, pp. 129–148.
- [20] B. Görer and F. B. Aydemir, "Generating requirements elicitation interview scripts with large language models," in *RE 2023 Workshops (REW)*. IEEE, 2023, pp. 44–51.
- [21] K. Ronanki, C. Berger, and J. Horkoff, "Investigating chatgpt's potential to assist in requirements elicitation processes," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2023, pp. 354–361.
- [22] B. Chen, K. Chen, S. Hassani, Y. Yang, D. Amyot, L. Lessard, G. Mussbacher, M. Sabetzadeh, and D. Varró, "On the use of gpt-4 for creating goal models: an exploratory study," in *RE 2023 Workshops*. IEEE, 2023, pp. 262–271.
- [23] S. de Kinderen and K. Winter, "Towards taming large language models with prompt templates for legal grl modeling," in *BPMDS 2024*. Springer, 2024, pp. 213–228.
- [24] N. Feng, L. Marso, S. G. Yaman, I. Standen, Y. Baartartogtokh, R. Ayad, V. O. De Mello, B. Townsend, H. Bartels, A. Cavalcanti et al., "Normative requirements operationalization with large language models," in *RE 2024*. IEEE, 2024, pp. 129–141.
- [25] A. Fantechi, S. Gnesi, L. Passaro, and L. Semini, "Inconsistency detection in natural language requirements using chatgpt: a preliminary evaluation," in *RE 2023*. IEEE, 2023, pp. 335–340.
- [26] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," in *ICSE 2023*. IEEE, 2023, pp. 1355–1367.
- [27] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *ICSE 2024*, 2024, pp. 1–13.
- [28] M. Sun, Y. Yang, Y. Wang, M. Wen, H. Jia, and Y. Zhou, "Smt solver validation empowered by large pre-trained language models," in *ASE 2023*. IEEE, 2023, pp. 1288–1300.
- [29] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," in *ICSE 2024*, 2024, pp. 1–12.