LogiCase: Effective Test Case Generation from Logical Description in Competitive Programming

Sicheol Sung[1] , **Aditi**[2] , **Dogyu Kim**[3] , **Yo-Sub Han**[1] and **Sang-Ki Ko**[2*]
[1]Yonsei University
[2]University of Seoul
[3]Kangwon National University
{sicheol.sung, emmous}@yonsei.ac.kr, dogyu.kim9@gmail.com, {aditimzu16,sangkiko}@uos.ac.kr

Abstract

Automated Test Case Generation (ATCG) is crucial for evaluating software reliability, particularly in competitive programming where robust algorithm assessments depend on diverse and accurate test cases. However, existing ATCG methods often fail to meet complex specifications or generate effective corner cases, limiting their utility. In this work, we introduce Context-Free Grammars with Counters (CCFGs), a formalism that captures both syntactic and semantic structures in input specifications. Using a fine-tuned CodeT5 model, we translate natural language input specifications into CCFGs, enabling the systematic generation of high-quality test cases. Experiments on the CodeContests dataset demonstrate that CCFG-based test cases outperform baseline methods in identifying incorrect algorithms, achieving significant gains in validity and effectiveness. Our approach provides a scalable and reliable grammar-driven framework for enhancing automated competitive programming evaluations.

1 Introduction

Automated Test Case Generation (ATCG) [Anand *et al.*, 2013] is a critical component of software engineering, particularly in competitive programming, where the performance and correctness of algorithms are tested against diverse and challenging inputs. As the complexity of software and algorithmic problems increases, manual test case creation becomes infeasible, prompting the rise of automated solutions. Recent advancements in deep learning have significantly improved the generation of test cases, yet several fundamental challenges persist. Ensuring adherence to intricate input specifications and capturing corner cases effectively remain elusive, often necessitating additional manual analysis.

The inadequacy of test suites has been a persistent issue in software testing. For example, the Defects4J dataset, as highlighted by Fraser and Arcuri [2011], has shown that incomplete test cases lead to insufficient program analysis. Similarly, the CodeNet dataset, a benchmark for competitive programming [Puri *et al.*, 2021], suffers from a lack of high-quality test cases, as noted by Zhao *et al.* [2024]. These limitations not only hinder robust algorithm validation but also exacerbate challenges in program repair [Tian *et al.*, 2022]. Even recent prompt-based methods leveraging Large Language Models (LLMs), such as ChatGPT, and mutation-based frameworks like MuTAP [Dakhel *et al.*, 2024], often produce invalid or incomplete test cases for competitive programming due to the complexity of test case specifications.

These shortcomings highlight a critical gap in automated testing: existing methods struggle to generate test cases that adhere strictly to complex problem constraints while capturing edge cases essential for algorithm validation. For instance, inaccuracies in some program synthesis benchmarks, as revealed by Liu *et al.* [2023], further emphasize the necessity of robust testing frameworks capable of addressing these gaps.

To tackle these challenges, we propose Context-Free Grammars with Counters (CCFGs), a novel formalism that integrates both syntactic and semantic elements of problem input specifications. By leveraging a fine-tuned CodeT5 [Wang *et al.*, 2021] model, our approach translates natural language descriptions into CCFGs, enabling the automated generation of test cases that are valid, specification-compliant, and highly effective in uncovering algorithmic flaws. Figure 1 illustrates our proposed framework.

Our main contributions are threefold:

- We introduce CCFGs to encode problem input specifications, capturing both syntax and semantics in a unified framework tailored for competitive programming.
- We develop a specialized model to map natural language problem descriptions to CCFGs, ensuring precise and specification-compliant test case generation.
- Using the CodeContests dataset [Li *et al.*, 2022], we demonstrate that CCFG-based test cases significantly outperform baseline methods in identifying algorithmic errors and distinguishing correct from incorrect solutions.

Note that the CodeContests dataset is a powerful resource for benchmarking and evaluating test case generation methods due to its diverse collection of competitive programming platforms such as AtCoder, CodeChef, Codeforces, etc.

One of the key strengths of our approach lies in its ability to overcome the primary limitation of grammar-based test case generation techniques—the need for a manually written input grammar for each specification. Unlike traditional methods

[*] Corresponding author

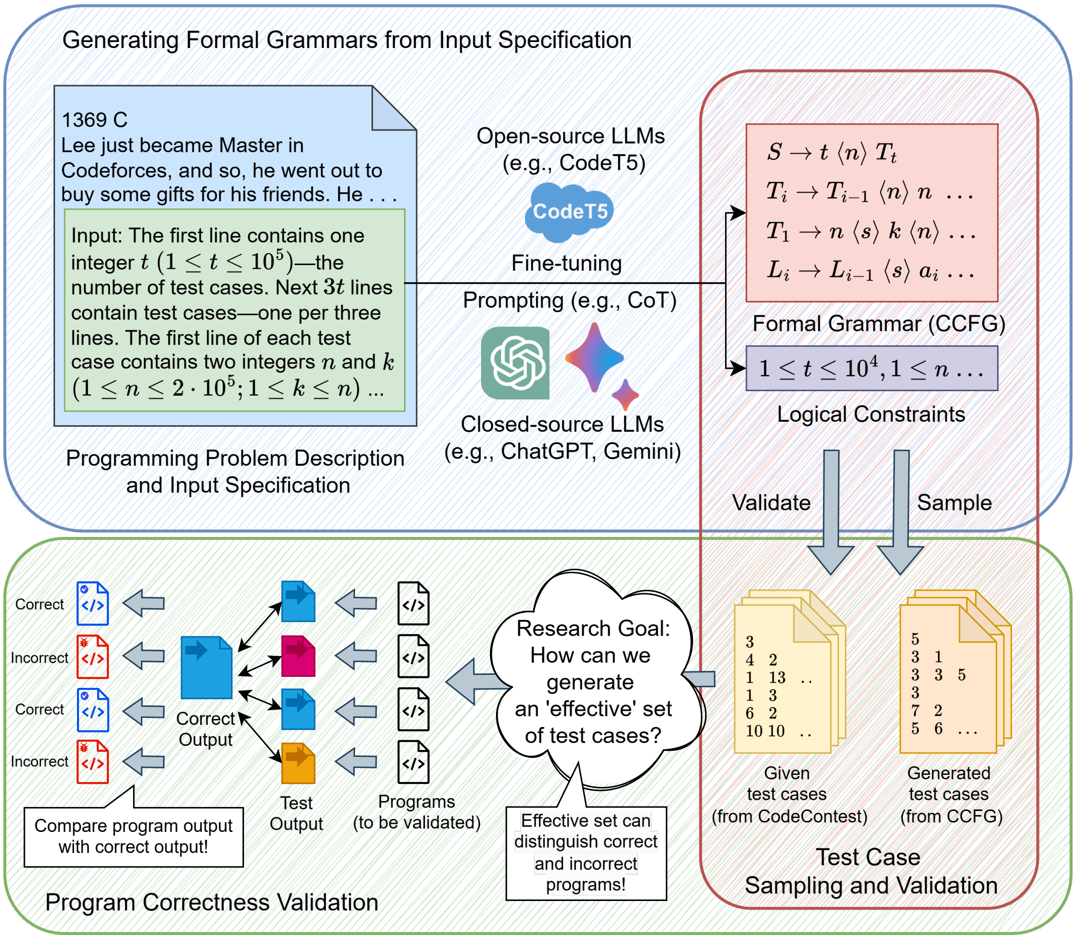
Generating Formal Grammars from Input Specification
1369 C
Lee just became Master in Codeforces, and so, he went out to buy some gifts for his friends. He . . .
Input: The first line contains one integer t $(1 \leq t \leq 10^5)$—the number of test cases. Next $3t$ lines contain test cases—one per three lines. The first line of each test case contains two integers n and k $(1 \leq n \leq 2 \cdot 10^5; 1 \leq k \leq n)$ ...
Programming Problem Description and Input Specification
Open-source LLMs (e.g., CodeT5)
CodeT5
Fine-tuning
Prompting (e.g., CoT)
Closed-source LLMs (e.g., ChatGPT, Gemini)
$S \to t \langle n \rangle T_t$
$T_i \to T_{i-1} \langle n \rangle n \ \ldots$
$T_1 \to n \langle s \rangle k \langle n \rangle \ldots$
$L_i \to L_{i-1} \langle s \rangle a_i \ldots$
Formal Grammar (CCFG)
$1 \leq t \leq 10^4, 1 \leq n \ldots$
Logical Constraints
Validate
Sample
Correct
Incorrect
Correct
Incorrect
Correct Output
Test Output
Programs (to be validated)
Compare program output with correct output!
Research Goal: How can we generate an 'effective' set of test cases?
Effective set can distinguish correct and incorrect programs!
3 4 2 1 13 .. 1 3 6 2 10 10 ..
5 3 1 3 3 5 3 7 2 5 6 ...
Given test cases (from CodeContest)
Generated test cases (from CCFG)
Test Case Sampling and Validation
Program Correctness Validation

Figure 1: Overview of the proposed framework for generating test cases for competitive programming problems. The deep learning model translates specifications into CCFGs while preserving their meaning. Subsequently, the CCFGs are utilized to generate test cases.

that require crafting a dedicated grammar for every input specification, our approach generalizes seamlessly across all input specifications in competitive programming problems. This eliminates the need for extensive manual effort and expertise to define grammars, enabling broader applicability and reducing the risk of incomplete or erroneous test coverage. By bypassing the reliance on handcrafted grammars, our method ensures scalability and efficiency, adapting to diverse input formats without requiring tailored adjustments for each new specification. This capability not only simplifies the testing process but also enhances its robustness and versatility, making it particularly suited for dynamic and varied environments like competitive programming.

2 Related Work

2.1 Test Case Generation using LLMs

Recent advancements in LLMs have opened new possibilities for ATCG by leveraging their ability to understand and generate structured data, natural language, and code with remarkable accuracy [Feng *et al.*, 2020; Guo *et al.*, 2022; Li *et al.*, 2022; Rozière *et al.*, 2023; Team *et al.*, 2024].

TestAug [Yang *et al.*, 2022] and ChatTESTER [Yuan *et al.*, 2023] utilized ChatGPT to derive test cases directly from natural language descriptions of code, often accompanied by code snippets. These methods provide a straightforward way to automate test case generation but face limitations in covering complex program logic and edge cases due to the inherent difficulty in aligning natural language prompts with precise program semantics. TestEval [Wang *et al.*, 2024] introduced a dataset of 210 Python programs sourced from LeetCode and designed three evaluation tasks: overall coverage, targeted line/branch coverage, and targeted path coverage.

CodaMosa [Lemieux *et al.*, 2023] utilizes a programming dataset to generate test cases by leveraging the LLM (OpenAI's Codex [Chen *et al.*, 2021]). Codex produces initial test cases that are syntactically and semantically meaningful, providing a strong foundation for further exploration. These test cases are then iteratively mutated to maximize code coverage. However, while CodaMosa effectively optimizes for code coverage, this metric does not always correlate with fault detection or real-world program behavior.

Recently, Xia *et al.* [2024], introduced Fuzz4All generating test cases of many different languages by leveraging the multilingual capabilities of LLMs. This approach accepts inputs in various formats, such as example code snippets, program specifications, or documentation, and uses multiple prompts to guide the LLM in generating fuzzing inputs.

2.2 Grammar-based Test Case Generation

Grammar-based fuzzing is well-established in software testing that uses formal grammars to guide the generation of inputs. The foundational work on fuzz testing by Miller *et al.* [1990] laid the groundwork for input generation techniques, demonstrating the effectiveness of random input generation. However, random fuzzing techniques often fail when targeting structured inputs because they are unlikely to produce valid test cases. On the other hand, grammar-based fuzzing, which relies on CFGs to generate syntactically valid test inputs, marked a substantial improvement over purely random input generation techniques by enabling more targeted testing.

Grammar-based white-box fuzzing [Godefroid *et al.*, 2008] enhances traditional grammar-based fuzzing by incorporating symbolic execution, enabling the exploration of program paths. Srivastava and Payer [2021] have proposed Gramatron, which uses *grammar automatons* in conjunction with aggressive mutation operators to synthesize bug triggers faster.

Note that a significant limitation of grammar-based test case generation or fuzzing techniques is the requirement for a manually written input grammar. This dependency introduces several challenges, particularly in terms of time, expertise, and coverage. Crafting a comprehensive input grammar requires a deep understanding of the input structure, including syntax and semantics, which demands significant effort and domain-specific knowledge from the developer or tester.

3 Methodology

We introduce the Context-Free Grammars With Counters (CCFG) to mitigate the limitations of previous approaches to automated testing of competitive programming. CCFGs are tailor-made to capture the formal semantics of most programming competition problems in the concise form of formal grammar by utilizing the counters to capture numerical values specified in the specifications.

3.1 Context-Free Grammars With Counters

Let us start with a motivating example. Example 1 is an input specification of “1369_C. RationalLee” from Codeforces.[1]

Example 1 (Input Specification)**.**

- *The first line contains one integer* t.
- *Next* $3t$ *lines contain test cases—one per three lines.*
- *The first line of each test case contains two integers* n *and* k.
- *The second line of each test case contains* n *integers* $a_1, \ldots, a_n$.
- *The third line contains* k *integers* $b_1, \ldots, b_k$.

To explain the necessity of CCFGs, let us describe the input specification of Example 1 using a CFG as follows; for simplicity, we assume we modify a generation algorithm of CFG to sample values of variables t, n, k, a_i and b_i according to their constraints during the generation.

Example 2 (Incorrect CFG of Example 1)**.**

$$\begin{aligned}
&S \rightarrow t \text{ <n> } T,\\
&T \rightarrow T \text{ <n> } n \text{ <s> } k \text{ <n> } L \text{ <n> } Z\\
&\quad \mid\ n \text{ <s> } k \text{ <n> } L \text{ <n> } Z,\\
&L \rightarrow L \text{ <s> } a_i \mid a_i,\\
&Z \rightarrow Z \text{ <s> } b_i \mid b_i.
\end{aligned}$$

While the above grammar can generate all valid test cases, it also generates invalid ones. During generation, each application of the production rule for T produces a single test case. Therefore, we need to limit the number of such applications to t, where the value of t is determined during the generation.

Intuitively, CCFG restricts the number of applications of production rules by using a *counter* to track the number of applications and selecting the next production rule based on the counter's value. Using this approach, we can modify the CFG of Example 2 to obtain the correct CCFG in Example 3.

Example 3 (Correct CCFG of Example 1)**.**

$$\begin{aligned}
&S \rightarrow t \text{ <n> } T_t,\\
&T_i \rightarrow T_{i-1} \text{ <n> } n \text{ <s> } k \text{ <n> } L_n \text{ <n> } Z_k,\\
&T_1 \rightarrow n \text{ <s> } k \text{ <n> } L_n \text{ <n> } Z_k,\\
&L_i \rightarrow L_{i-1} \text{ <s> } a_i, \quad L_1 \rightarrow a_1,\\
&Z_i \rightarrow Z_{i-1} \text{ <s> } b_i, \quad Z_1 \rightarrow b_1.
\end{aligned}$$

Next, we generate test cases using the CCFG as follows. When applying the production rule for T_i to T_t, we set the value of an internal counter to t. Then, during the derivation of T_{i-1}, we decrement the counter by 1. Finally, when the counter value reaches 1, we apply the production for T_1 to T_i.

Note that the proposed CCFG is substantially different from the grammar studied by Chistikov *et al.* [2018] as our CCFG can associate the integer input with non-terminals by predefined production rules during the parsing while Chistikov *et al.* consider a grammar that resets or adds integers to a pre-defined set of counters at each step.

3.2 CCFGT5 Translation Model

Inspired by the idea of *grammar prompting* [Wang *et al.*, 2023], which enables LLMs to use domain-specific constraints expressed through a formal grammar, we propose a translation model, CCFGT5, designed to translate a problem input specification in NL into a concise CCFG while preserving the original semantics. This model incorporates two specifically fine-tuned CodeT5 modules: one focuses on grammar, and the other on constraints. We use Adam optimizer with learning rate 10^{-5} and cross-entropy loss function to train each CodeT5 model. We generate candidate grammars and constraints with repetition penalty 2.5 and length penalty 1.0 from each model.

We also use a specialized CCFG tokenizer to enable effective training. Our tokenizer converts a CCFG into a list of grammar symbols and labels each symbol type with descriptive words, such as 'variable' and 'nonterminal,' to enhance the readability from the LLM's perspective.

4 Experiments

We evaluate the practical usefulness of CCFGs through experimental validation. All implementations and associated codes

[1] https://codeforces.com/problemset/problem/1369/C

Category	Specification difficulty (#)			Total
	Easy	Normal	Hard	
Train	518	553	129	1,200
Evaluation	159	101	11	271

Table 1: Distribution of difficulty in the dataset. We exclude 29 problems from the test dataset that (1) lack incorrect solutions or (2) our implementation cannot process their human-labeled grammars.

and datasets used in these experiments are available in our GitHub repository.[2]

4.1 Dataset

We use the CodeContests dataset, which consists of various programming problems sourced from different competitive platforms [Li *et al.*, 2022]. This dataset includes algorithms for programs in various programming languages and *public*, *private*, and *generated* test cases for each problem.

We manually created CCFGs for 1,500 different problems based on their descriptions and categorized the human-labeled grammars into three levels: *easy*, *normal* and *hard*. Easy problems consist of simple grammars that consist solely of variables, without any complex structures or additional elements. Normal problems include grammars with one nonterminal that requires counting or tracking. Lastly, hard problems encompass grammars that have more than one nonterminal. These grammars are the most complex, involving multiple non-terminal elements and adding layers of complexity and structure. After categorizing the grammars, we split them into a training dataset with 1,200 problems and an evaluation dataset with 300 problems. Notably, the training dataset contains more difficult grammars to help the model learn complex syntactic structures effectively. Table 1 summarizes the difficulty distribution of each dataset.

We filter out 6 problems from the evaluation dataset for which CodeContests has no incorrect solutions that are necessary for evaluation. Additionally, we exclude 23 more problems that our CCFG implementation cannot fully support, due to the combinatorial complexity of input specification.

4.2 Test Case Sampling

As in Example 1, the value of a variable in a test case often determines the number of subsequent lines or variables in that test case. Therefore, we can control the length of the generated test cases by varying the interval of variable sampling during the generation. Instead of using the original interval $(n, n+k)$, we sample a value of a variable from one of the following options: (1) the interval $(n, n+k)$, (2) the interval $(n, n+\log k)$, (3) $(n, n+\log\log k)$, or (4) the minimum value n. When generating ten *long* test cases, we first create a test case using option (1). If it fails (e.g., due to the timeout), we then try the options (2) and (3) in order. We also generate ten *medium* and *short* test cases by starting with options (2) and (3), respectively. Finally, if we succeed in generating a corner case with option (4), we replace one of the short test cases with the corner case.

4.3 Baselines

Mutation-based fuzzing. We utilize the public and private test cases from the CodeContests dataset, tokenizing these test cases based on spaces and newline characters. We then randomly select 30% of tokens for mutation, adapting our approach according to the token type: integer, float, or string. This selective mutation process enables effective fuzzing while still adhering to the original input specifications.

Direct test case generation from LLMs. We employ two LLMs, OpenAI's ChatGPT 4 and Google's Gemini, to generate test cases directly. We provide an input specification and a strict format required for generating the test cases. To fully exploit the performance of LLMs, we use the Chain-of-Thought (CoT) [Wei *et al.*, 2022] style prompt.

4.4 Evaluation Metrics

We carefully design evaluation metrics to address the following research questions throughout experimental results: (1) Is the CCFG-based approach better than direct test case generation? (2) Which method is most effective for generating CCFGs from descriptions?

Validity and generality. We say that a test case is *valid* if it follows the logical input specification of the problem. We evaluate the validity of each test case by determining whether or not the ground-truth grammar can parse the test case. Since it is computationally undecidable [Hopcroft *et al.*, 2007] to decide whether or not a given CCFG is valid, we instead empirically measure *element-based validity* of grammar as the ratio of valid test cases to the total number of test cases. Additionally, we measure *set-based validity* of a grammar by checking whether all generated test cases are valid. The value of the set-based validity is either 0 or 1, while element-based validity can take any value in between. We say a set of test cases is *valid* if its set-based validity is 1.

Note that the validity alone cannot ensure that the grammar generates all the possible test cases described by the input specification. To measure how many valid test cases can be covered by a grammar, we define *element-based generality* of a grammar as the ratio of test cases that can be parsed by the grammar to the total number of test cases generated by the ground-truth grammar. A *set-based generality* of a grammar is 1 if and only if the test case-based generality is also 1. In this case, we call the grammar is *general*; otherwise, the set-based generality is 0. If the set of test cases generated by a grammar is valid and the grammar is general, we say that the grammar is (empirically) *semantically equivalent* to the ground-truth grammar. In contrast, we say that a grammar is *syntactically equivalent* to the ground truth if two grammars are equivalent except for the naming of variables.

Effectiveness. The primary purpose of test cases in competitive programming is to distinguish between correct and incorrect algorithms. For a given problem p, let A_p be a set of all incorrect algorithms implying that for each $y \in A_p$, there always exists a valid test case x such that $\hat{y}(x) \neq y(x)$, where $\hat{y}(x)$ is the correct output for x. Then, we define the *effectiveness* $E(x, A_p)$ of a test case x with respect to A_p as

$$E(x, A_p) := \frac{|\{y \in A_p \mid y(x) \neq \hat{y}(x)\}|}{|A_p|},$$

[2] https://github.com/Aditi1612/Grammar-based-test-case-generation

Category	**Method**	**Well-defined** (#)	**Validity** (%)		**Effectiveness** (%)	
			Element-based	Set-based	Element-based	Set-based
CodeContests	Public	270	99.63	99.63	31.71	39.63*
	Private	208	76.75	76.75	39.93	70.89*
	Generated	269	77.95	30.26	18.25	28.08*
Direct gen.	Gemini	271	80.15	61.25	26.62	44.18
	ChatGPT	271	**91.38**	78.97	37.75	63.54
Fuzzing	Public	270	79.26	45.02	17.40	28.77*
	Private	208	63.65	33.58	17.59	28.16*
CCFG-based	Gemini$_1$	108	16.06	15.50	8.06	13.14
	Gemini$_5$	158	46.04	44.65	24.02	39.74
	ChatGPT$_1$	211	67.15	66.42	32.16	54.04
	ChatGPT$_5$	226	78.43	77.86	37.81	65.29
	CcfgT5$_1$	132	19.57	18.82	10.08	15.90
	CcfgT5$_{10}$	264	82.32	**81.18**	**42.26**	**67.73**
CCFG-based	Ground-truth	271	100.00	100.00	52.51	83.40

Table 2: Validity and effectiveness of the test cases across different methods. CcfgT5$_n$ refers the grammars generated by CcfgT5 model with beam-size n. We select a pair of grammar and constraints among top-k grammars and constraints. Gemini-n and ChatGPT-n refer to the CCFGs produced by LLMs employing CoT with n different examples. Note that set-based effectiveness for both the CodeContests and Fuzzing categories, marked with an asterisk (*), may involve more or fewer than 10 test cases for each problem, which limits the comparability of these results with the set-based effectiveness from other methods.

which is the ratio of incorrect algorithms in A_p that are *distinguishable* by the test case x to all incorrect algorithms A_p. We extend to define the *element-based effectiveness* $E_{\text{elt}}(X, A_p)$ of a set X of test cases with respect to A_p as

$$E_{\text{elt}}(X, A_p) := \begin{cases} \frac{1}{|X|}\sum_{x\in X} E(x, A_p), & \text{if } X \text{ is valid,} \\ 0, & \text{otherwise.} \end{cases}$$

Note that the element-based effectiveness of a set of test cases is the average effectiveness of individual test cases in the set. If each test case can distinguish different incorrect algorithms, then the entire set can identify more incorrect algorithms compared to any single test case. Therefore, the diversity of the algorithms that the set can distinguish is also important. Thus, we define *set-based effectiveness* $E_{\text{set}}(X, A_p)$ of a set X of test cases as

$$\begin{aligned} &E_{\text{set}}(X, A_p) \\ &:= \begin{cases} \dfrac{|\{y \in A_p \mid y(x) \neq \hat{y}(x), \exists x \in X\}|}{|A_p|}, & \text{if } X \text{ is valid,} \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

For experiments, we determine the correct output $\hat{y}(x)$ for a test case x by executing up to ten correct algorithms from the dataset and selecting the most frequently occurring output as the correct one. Additionally, for each problem p, we sample at most ten incorrect algorithms to create a set designated as A_p. We treat the output of algorithms that exceed twice the original timeout as $\perp$, which is always considered incorrect.

We use a total of ten test cases from thirty test cases generated by CCFGs to compute set-based effectiveness, consisting of four short test cases (including corner case if possible) and three medium and long test cases. This approach ensures that we are using the same number of test cases as in the case of direct generation.

4.5 Analysis of Experimental Results

Table 2 presents statistics for test cases generated by either baseline algorithms or CCFGs. CCFG-based test cases with CcfgT5$_{10}$ exhibit the highest set-based validity and both types of effectiveness, achieving 81.18%, 42.26% and 67.73%. In contrast, direct generation using ChatGPT shows the highest element-based validity with 91.38%. Furthermore, when comparing the results of direct generation of ChatGPT and those of CCFG-based ChatGPT$_5$, the latter shows higher effectiveness (65.29% compared to 63.54%).

It is important to note that the differences between the element-based validity and the set-based validity of CCFG-based approaches are relatively smaller than those in other categories. This suggests that the validity of each test case generated by the CCFG-based approach is more consistent for each problem specification. Once correct grammar and constraints are established, only valid test cases are generated. Additionally, when considering ChatGPT, the set-based validity over well-defined test sets with the CCFG-based approach is 93.36% (77.86% × $271/226$), which is higher than the validity of direct generation at 78.97%. This indicates that the CCFG-based approach is more likely to fail in generating test cases than producing invalid ones.

As mentioned in Section 4.1, we use a human-labeled grammar as the ground-truth only when the grammar can parse all the public and private test cases in the CodeContests dataset. Therefore, the validity of these test cases is represented by the ratio of problems with well-defined test case sets to the number

Method	Well-defined (#)			Set-based validity (%)			Set-based effectiveness (%)		
	Easy	Normal	Hard	Easy	Normal	Hard	Easy	Normal	Hard
Gemini$_{\text{(Direct)}}$	159	101	11	71.07	47.52	45.45	53.49	31.55	25.45
ChatGPT$_{\text{(Direct)}}$	159	101	11	**88.68**	65.35	63.64	**74.37**	47.57	53.64
Gemini$_5$	97	55	6	47.17	41.58	36.36	42.58	36.14	31.82
ChatGPT$_5$	138	80	8	83.02	72.28	54.55	70.69	58.54	49.09
CCFGT5$_{10}$	156	97	11	83.65	**76.24**	**90.91**	68.47	**65.32**	**79.09**
Ground-truth	159	101	11	100.00	100.00	100.00	81.51	85.67	90.00

Table 3: Validity and effectiveness with respect to the difficulty of input specifications for problems. The first two rows present results for the direct test case generation approach, while the remaining rows are for the CCFG-based approach.

of total problems: 76.75% (208 out of 271) and 99.63% (270 out of 271). We also anticipate that the element-based and set-based effectiveness could reach 52.51% and 83.40%, respectively, which are of ground-truth grammars, if the model successfully generates semantically correct grammars.

Observation 1: CCFGs are especially effective for problems with more complex input specifications.

Table 3 shows the relationship between the difficulty of test case specifications, set-based validity, and effectiveness. We observe that the direct test case generation using ChatGPT achieves the highest validity and effectiveness, with 88.68% and 74.37% for problems with easy specifications. In contrast, CCFGT5$_{10}$ shows the highest results for problems with normal or hard specifications.

Interestingly, CCFGT5$_{10}$ exhibits a less pronounced tendency for validity to decrease as difficulty increases compared to other methods. This observation also highlights the robustness of CCFGT5 in understanding complex specifications.

Observation 2: Longer test cases tend to be more (element-wise) effective, but the set of test cases with various lengths is the most effective.

Table 4 presents the relationship between the length of test cases and effectiveness. This result indicates that the variation in intervals during generation impacts the efficiency of the resulting test case set, suggesting there is potential for increasing efficiency by improving the generation algorithm of CCFG. We also observe that the union of short and long test cases exhibits the highest effectiveness among combinations from the results using 20 test cases. This indicates that short and long test cases identify different incorrect problems.

Consequently, we conclude that utilizing a CCFG-based approach not only eliminates the need for individual test case validation but also enhances the effectiveness of test cases, particularly when input specifications are complex. Additionally, there is potential to further increase effectiveness by generating more test cases, which is straightforward with CCFGs, along with improved heuristics for diverse generation.

Observation 3: CCFGT5 generates more valid and general grammars than LLMs such as ChatGPT and Gemini.

The evaluation results presented in Table 5 demonstrate CCFGT5 produces the most general grammars (81.18%) and those that are semantically equivalent (78.23%). Notably, the average element-based generality closely aligns with the average set-based generality across all methods, with differences of less than or equal to 4.15%p for ChatGPT$_5$. Since this difference reflects the ratio of ground-truth test cases parsed by non-general grammars, it suggests that determining whether a grammar is general using only a few sampled ground-truth test cases is reliable. When we compare CCFGT5 and LLMs, the ratios of syntactically different grammars to semantically equivalent grammars are 42.63% ($1-16.24/40.96$) and 63.68% ($1-49.82/78.23$) for GPT and CCFGT5, respectively. It indicates that LLMs generate more syntactically variant grammars compared to the fine-tuned CodeT5 model. Additionally, we use Jaccard similarity $J(G,V)$, which represents the ratio of the intersection to the union to measure the similarity between the set G of general grammars and the set of valid grammars V:

$$J(G,V) = \frac{\text{(Semantic Equality)} \times 100(\%)}{\text{(Set-General.)} + \text{(Set-Valid.)} - \text{(Sem. Equ.)}}.$$

As a result, all similarities are greater than 80%, with CCFGT5$_{10}$ showing the largest similarity with 92.98%. This indicates that the most valid grammars are also general, and they can generate all the test cases based on how we sample the variables.

4.6 Case Study

We analyzed several failure cases where our proposed CCFGT5 model struggles to generate accurate CCFGs and their associated constraints. Frequently, the model misinterprets natural language constraints, converting them incorrectly into numerical constraints, and often handles grammars in natural language format rather than structured CCFG. This leads to incomplete coverage of specified constraints and the omission of additional implicit constraints. Also, the model fails to cover all the constraints present in the CCFG, along with some unknown constraints that are not included in the problem specification.

Method	**Set-based effectiveness w/ 10 test cases**				**w/ 20 test cases**			**w/ 30 test cases**
	Short	Medium	Long	Mixed	S+M	S+L	M+L	Short+Medium+Long
Gemini$_5$	32.77	34.76	36.35	39.74	37.64	40.66	39.37	41.66
ChatGPT$_5$	53.51	57.25	60.34	65.29	62.29	67.66	65.38	68.62
CCFGT5$_{10}$	57.86	61.05	64.05	**67.73**	65.98	**71.27**	68.33	**72.04**
Ground-truth	71.06	73.14	78.20	83.40	80.61	86.75	83.33	88.12

Table 4: Set-based effectiveness with respect to the length of test cases. For the columns of S+M, S+L, and M+L, we used the union of two sets selected from the of short (S), medium (M) and long (L) test cases. Note that the fifth column (Mixed) uses the same sets of test cases as in Table 2, which consists of 4 short, 3 medium and 3 long test cases.

Method	**Generality** (%)		**Validity** (%)	**Equality** (%)	
	Element-based	Set-based	Set-based	Semantic	Syntactic
Gemini$_1$	14.18	13.28	15.50	12.92	3.69
Gemini$_5$	42.84	41.70	44.65	40.96	16.24
ChatGPT$_1$	59.87	55.72	66.42	54.61	14.39
ChatGPT$_5$	72.93	71.22	77.86	70.11	29.89
CCFGT5$_1$	19.95	19.56	18.82	18.08	11.07
CCFGT5$_{10}$	**83.44**	**81.18**	**81.18**	**78.23**	**49.82**

Table 5: Generality and semantic equality of generated grammars to ground-truth grammars. A generated grammar is semantically equivalent to the ground-truth grammar if and only if the grammar is valid and general.

In Example 4, the model extracts a constraint for a variable n in natural language rather than converting it in to the expected formal expression $n \leq 10^5$. Note that every constraint in ground-truth grammars consists solely of formal expressions.

Our model encounters difficulties in understanding the necessity of translating the natural language explanation into a formal representation. This failure is caused by the fact that most specifications in our training dataset contain explicit formal representations for constraints, while this example has such constraints in natural language.

Example 4 (139_D. Digit Permutations from Codeforces)**.**

```
The first line contains a positive
integer n --- the original number. The
number of digits in this number does not
exceed 10ˆ5. The number is written
without any leading zeroes.
```

```
Grammar: ["<S> -> n"]
Constraints:
["Thenumberdoesnotexceed10ˆ5."]
```

Additionally, in Example 5, CCFGT5 erroneously conflates `xi` as a variable that does not use a counter. This indicates that the model fails to understand the meaning of "`Each line`", which implies a use of a counter for `x`.

In contrast, the model accurately produces `x_i` as a variable that uses counters in `Constraint`. This inconsistency arises because we compose CCFGT5 with two independent fine-tuned CodeT5 modules for grammars and constraints, which assign different meanings for `xi`.

Example 5 (103_C. Russian Roulette from Codeforces)**.**

```
The first line contains three integers n,
k and p (...) Then follow p lines; they
are the queries. Each line contains one
integer xi (1 <= xi <= n) (...)
```

```
Grammar: [
  ...,
  "<Y_i> -> <Y_i-1> <n> xi",
  "<Y_1> -> xi"
]
Constraints: [..., "1<=x_i<=n"]
```

These failures are caused by uncommon natural language expressions for constraints and inconsistent variable representations in the CodeContests dataset. Data augmentation and normalization can address these issues.

5 Conclusions

We presented a novel framework leveraging CCFGs for ATCG in competitive programming. By translating input specifications into formal grammars, our method bridges the gap between specification complexity and test case validity, offering substantial improvements in the accuracy and coverage of generated test cases. Experiments highlight the effectiveness of our approach in distinguishing incorrect algorithms and ensuring specification compliance.

We will expand CCFGs to handle broader input domains and optimizing sampling strategies. By advancing these directions, our methodology could serve as a foundation for scalable, reliable test case generation across diverse software engineering applications.

Acknowledgments

Sung and Han were supported by the NRF grant (RS-2025-00562134) and the AI Graduate School Program at Yonsei University (RS-2020-II201361) funded by the Korean government (MSIT). Aditi and Ko were supported by the NRF grant (RS-2023-00208094) funded by MSIT.

References

[Anand *et al.*, 2013] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001, 2013.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[Chistikov *et al.*, 2018] Dmitry Chistikov, Christoph Haase, and Simon Halfon. Context-free commutative grammars with integer counters and resets. *Theoretical Computer Science*, 735:147–161, 2018.

[Dakhel *et al.*, 2024] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468, 2024.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics*, volume EMNLP 2020, pages 1536–1547. Association for Computational Linguistics, 2020.

[Fraser and Arcuri, 2011] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference*, pages 416–419. ACM, 2011.

[Godefroid *et al.*, 2008] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215. ACM, 2008.

[Guo *et al.*, 2022] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 7212–7225. Association for Computational Linguistics, 2022.

[Hopcroft *et al.*, 2007] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley, 2007.

[Lemieux *et al.*, 2023] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering*, pages 919–931. IEEE, 2023.

[Li *et al.*, 2022] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378:1092–1097, 2022.

[Liu *et al.*, 2023] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

[Miller *et al.*, 1990] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[Puri *et al.*, 2021] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[Rozière *et al.*, 2023] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023.

[Srivastava and Payer, 2021] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 244–256. ACM, 2021.

[Team *et al.*, 2024] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. Gemma: Open models based on gemini research and technology. *CoRR*, abs/2403.08295, 2024.

[Tian *et al.*, 2022] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. Predicting patch correctness based on the similarity of failing test cases. *ACM*

Transactions on Software Engineering and Methodology, 31(4):77:1–77:30, 2022.

[Wang *et al.*, 2021] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *CoRR*, abs/2109.00859, 2021.

[Wang *et al.*, 2023] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems*, 2023.

[Wang *et al.*, 2024] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. TESTEVAL: benchmarking large language models for test case generation. *CoRR*, abs/2406.04531, 2024.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny 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.

[Xia *et al.*, 2024] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the 46th International Conference on Software Engineering*, pages 126:1–126:13. ACM, 2024.

[Yang *et al.*, 2022] Guanqun Yang, Mirazul Haque, Qiaochu Song, Wei Yang, and Xueqing Liu. TestAug: A framework for augmenting capability-based NLP tests. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 3480–3495. International Committee on Computational Linguistics, 2022.

[Yuan *et al.*, 2023] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving ChatGPT for unit test generation. *CoRR*, abs/2305.04207, 2023.

[Zhao *et al.*, 2024] Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. RePair: Automated program repair with process-based feedback. In *Findings of the Association for Computational Linguistics*, pages 16415–16429, 2024.