Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization

Pedro Orvalho[1]*, **Mikoláš Janota**[2], **Vasco M. Manquinho**[3]
[1]Department of Computer Science, University of Oxford, Oxford, UK
[2]CIIRC, Czech Technical University in Prague, Czechia
[3]INESC-ID, IST, Universidade de Lisboa, Portugal
pedro.orvalho@cs.ox.ac.uk, mikolas.janota@cvut.cz, vasco.manquinho@tecnico.ulisboa.pt

Abstract

Automated Program Repair (APR) for introductory programming assignments (IPAs) is motivated by the large number of student enrollments in programming courses each year. Since providing feedback on programming assignments requires substantial time and effort from faculty, personalized automated feedback often involves suggesting repairs to students' programs. Symbolic semantic repair approaches, which rely on Formal Methods (FM), check a program's execution against a test suite or reference solution, are effective but limited. These tools excel at identifying buggy parts but can only fix programs if the correct implementation and the faulty one share the same control flow graph. Conversely, Large Language Models (LLMs) are used for program repair but often make extensive rewrites instead of minimal adjustments. This tends to lead to more invasive fixes, making it harder for students to learn from their mistakes. In summary, LLMs excel at completing strings, while FM-based fault localization excel at identifying buggy parts of a program.

In this paper, we propose a novel approach that combines the strengths of both FM-based fault localization and LLMs, via zero-shot learning, to enhance APR for IPAs. Our method uses MaxSAT-based fault localization to identify buggy parts of a program, then presents the LLM with a program sketch devoid of these buggy statements. This hybrid approach follows a Counterexample Guided Inductive Synthesis (CEGIS) loop to iteratively refine the program. We ask the LLM to synthesize the missing parts, which are then checked against a test suite. If the suggested program is incorrect, a counterexample from the test suite is fed back to the LLM for revised synthesis. Our experiments on 1,431 incorrect student programs show that our counterexample guided approach, using MaxSAT-based bug-free program sketches, significantly improves the repair capabilities of all six evaluated LLMs. This method allows LLMs to repair more programs and produce smaller fixes, outperforming other configurations and state-of-the-art symbolic program repair tools.

Code — https://doi.org/10.5281/zenodo.14517771

Introduction

Every year, thousands of students enroll in programming-oriented courses. With the rapid growth of Computer Science courses, providing personalized and timely feedback on *introductory programming assignments* (IPAs) and software projects has become a significant challenge, requiring substantial time and effort from faculty (Orvalho, Janota, and Manquinho 2024c, 2022b).

Automated Program Repair (APR) has emerged as a promising solution to this challenge, aiming to deliver automated, comprehensive, and personalized feedback to students about their programming errors (Gulwani, Radicek, and Zuleger 2018; Ahmed et al. 2022; Wang, Singh, and Su 2018; Hu et al. 2019). Traditional semantic APR techniques based on *Formal Methods* (FM), while providing high-quality fixes, are often slow and may struggle when the correct implementation diverges significantly from the erroneous one (Contractor and Rivero 2022). These APR approaches do not guarantee minimal repairs, as they align an incorrect submission with a correct implementation for the same IPA. If the alignment is not possible, these tools return a structural mismatch error, leaving the program unrepaired. In the past decade, there has been a surge in Machine Learning (ML) techniques for APR (Gupta et al. 2017; Mesbah et al. 2019; Gupta, Kanade, and Shevade 2019; Yasunaga and Liang 2020; Rolim et al. 2017; Pu et al. 2016; Bhatia, Kohli, and Singh 2018; Orvalho et al. 2023). ML-based approaches require multiple correct implementations to generate high-quality repairs, and need considerable time and resources to train on correct programs. While these approaches generate repairs more quickly, they often produce imprecise and non-minimal fixes (Wang, Singh, and Su 2018).

More recently, *Large Language Models* (LLMs) trained on code (LLMCs) have shown great potential in generating program fixes (Joshi et al. 2023; Xia, Ding, and Zhang 2023; Jin et al. 2023; Wei, Xia, and Zhang 2023; Fan et al. 2023; Xia, Wei, and Zhang 2023; Zhang et al. 2024; Phung et al. 2023). LLM-based APR can be performed using zero-shot learning (Xia and Zhang 2022), few-shot learning (Zhang et al. 2024) or fine-tuned models (Jin et al. 2023). Fine-tuned models are the most commonly used, where the model is trained for a specific task. Conversely, zero-shot learning refers to the ability of a model to correctly perform a task without having seen any examples of that task during training. Few-shot learning refers to the LLMs's ability to perform tasks correctly with only a small number of examples provided. Furthermore, the ability to generalize using

*Part of this work was conducted at INESC-ID, IST, UL.

zero or few-shot learning enables LLMs to handle a wide range of tasks without the need for costly retraining or fine-tuning. Nonetheless, few-shot learning can lead to larger fixes than necessary, as it is based on a limited number of examples. LLMs do not guarantee minimal repairs and typically rewrite most of the student's implementation to fix it, rather than making minimal adjustments, making their fixes less efficient and harder for students to learn from.

In this paper, we propose a novel approach that combines the strengths of both FM and LLMs to enhance APR of IPAs via zero-shot learning. Our method involves using MaxSAT-based fault localization to identify the set of minimal buggy parts of a program and then presenting an off-the-self LLM with a program sketch devoid of these buggy statements. This hybrid approach follows a Counterexample Guided Inductive Synthesis (CEGIS) loop (Solar-Lezama et al. 2005) to iteratively refine the program. We provide the LLM with a bug-free program sketch and ask it to synthesize the missing parts. After each iteration, the synthesized program is checked against a test suite. If the program is incorrect, a counterexample from the test suite is fed back to the LLM, prompting a revised synthesis.

Our experiments with 1431 incorrect student programs reveal that our counterexample guided approach, utilizing MaxSAT-based bug-free program sketches, significantly boosts the repair capabilities of all six evaluated LLMs. This method enables LLMs to repair more programs and produce superior fixes with smaller patches, outperforming both other configurations and state-of-the-art symbolic program repair tools (Gulwani, Radicek, and Zuleger 2018; Ahmed et al. 2022).

In summary, this paper makes the following contributions:

- We tackle the Automated Program Repair (APR) problem using an LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach;
- We employ MaxSAT-based Fault Localization to guide and minimize LLMs' patches to incorrect programs by feeding them bug-free program sketches;
- Experiments show that our approach enables all six evaluated LLMs to fix more programs and produce smaller patches than other configurations and symbolic tools.

Motivation

Consider the program presented in Listing 1, which aims to determine the maximum among three given numbers. However, based on the input-output test suite $T = \{t_0 = ((1,2,3),3);\ t_1 = ((6,2,1),6);\ t_2 = ((-1,3,1),3);\}$, the program is buggy as its output differs from the expected results. The set of minimal faulty lines in this program includes lines 4 and 8, as these two `if` conditions are incorrect according to the test suite. A good way to provide personalized feedback to students on their IPAs is to highlight these two buggy lines. However, it is essential to check these faults by fixing the program and evaluating it against the test suite.

Using traditional Automated Program Repair (APR) tools for IPAs based on Formal Methods, such as CLARA (Gulwani, Radicek, and Zuleger 2018) or VERIFIX (Ahmed et al. 2022), the program in Listing 1 cannot be fixed within 90 seconds. CLARA takes too long to compute a 'minimal' repair by considering several correct implementations for the same IPA, while VERIFIX returns a compilation error. Conversely, using state-of-the-art LLMs trained for coding tasks (LLMCs), GRANITE (Mishra 2024) or CODEGEMMA (Zhao 2024), would involve providing the description of the programming assignment and some examples of input-output tests. Even with these features, neither LLM could fix the buggy program in Listing 1 within 90 seconds when repeatedly testing and refining their fixes. If the lecturer's reference implementation shown in Listing 2 is suggested as a reference in the prompt, both LLMs simply copy the correct program, ignoring instructions not to do so.

Hence, symbolic approaches demand an excessive amount of time to produce an answer, and LLMs, while fast, often produce incorrect fixes. A promising strategy to provide feedback to students on IPAs is to combine the strengths of both approaches. MaxSAT-based Fault localization (Jose and Majumdar 2011a; Ignatiev et al. 2019) can rigorously identify buggy statements, which can then be highlighted in the LLM prompt to focus on the specific parts of the program that need fixing. Listing 3 shows an example of a program sketch, which is a partially incomplete program where each buggy statement from the original incorrect program in Listing 1 is replaced with a `@ HOLE @`. Instructing the LLMs to complete this incomplete program allows both GRANITE and CODEGEMMA to fix the buggy program in a single interaction, returning the program in Listing 4.

Preliminaries

This section provides definitions used throughout the paper.

Synthesis Problem. For a given program's specification S (e.g., input-output examples), G a context-free grammar (CFG), and O be the semantics for a particular Domain-specific language (DSL), the goal of *program synthesis* is to infer a program $\mathcal{P}$ such that (1) the program is produced by G, (2) the program is consistent with O and (3) $\mathcal{P}$ is consistent with S (Orvalho et al. 2019).

Semantic Program Repair. Given (T, G, O, P), let T be a set of input-output examples (test suite), G be a grammar, O be the semantics for a particular Domain-specific language, and P be a syntactically well-formed program (i.e. sets of statements, instructions, expressions) consistent with G and O but semantically erroneous for at least one of the input-output tests i.e., $\exists\{t^i_{in}, t^i_{out}\} \in T\ :\ P(t^i_{in}) \neq t^i_{out}$. The goal of *Semantic Program Repair* is to find a program P_f by semantically change a subset S_1 of P's statements ($S_1 \subseteq P$) for another set of statements S_2 consistent with G and O, such that, $P_f = ((P \setminus S_1) \cup S_2)$ and $\forall\{t^i_{in}, t^i_{out}\} \in T\ :\ P_f(t^i_{in}) = t^i_{out}$.

Counterexample Guided Inductive Synthesis (CEGIS). CEGIS is an iterative algorithm commonly used in Program Synthesis and Formal Methods to construct programs or solutions that satisfy a given specification (Abate et al. 2018; Jha et al. 2010; Solar-Lezama et al. 2005). CEGIS consists of two steps: the synthesis step and the verification step. Given the specification of the desired program,

Listing 1 Semantically incorrect program. Faulty lines: {4,8}.

```
1  int main(){ // finds maximum of 3 numbers
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f<s && f>=t) //fix: f>=s
5        printf("%d",f);
6     else if (s>f && s>=t)
7        printf("%d",s);
8     else if (t<f && t<s) //fix: t>f and t>s
9        printf("%d",t);
10    return 0;
11 }
```

Listing 2 Reference implementation.

```
1  int main() {
2     int m1,m2,m3,m;
3     scanf("%d%d%d",&m1,&m2,&m3);
4     m = m1 > m2 ? m1 : m2;
5     m = m3 > m ? m3 : m;
6     printf("%d\n", m);
7     return 0;
8  }
```

Listing 3 Program sketch with holes.

```
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     @ HOLE 1 @
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     @ HOLE 2 @
9        printf("%d",t);
10    return 0;
11 }
```

Listing 4 GRANITE's fix using the program sketch.

```
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f >= s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else
9        printf("%d",t);
10    return 0;
11 }
```

the inductive synthesis procedure generates a candidate program. Next, the candidate program P is passed to the verification step, which checks whether P satisfies all possible inputs' specifications. Otherwise, the Decider produces a counterexample c from the satisfying assignment, which is then added to the set of inputs passed to the synthesizer, and the loop repeats. The synthesis engine refines its hypothesis using this counterexample to avoid similar mistakes in subsequent iterations. This iterative loop (comprising candidate generation, verification, counterexample generation, and refinement) continues until a correct candidate is found that satisfies all given specifications and constraints.

Maximum Satisfiability (MaxSAT). The *Boolean Satisfiability* (SAT) problem is the decision problem for propositional logic (Biere et al. 2009). A propositional formula in Conjunctive Normal Form (CNF) is a conjunction of clauses where each clause is a disjunction of literals. The *Maximum Satisfiability* (MaxSAT) problem is an optimization version of SAT, i.e., the goal is to find an assignment that maximizes the number of satisfied clauses in a CNF formula (Orvalho, Manquinho, and Martins 2023).

Formula-based Fault Localization (FBFL). Given a faulty program and a test suite with failing test cases, *formula-based fault localization* (FBFL) methods encode the localization problem into an optimization problem to identify a minimal set of faulty statements (diagnoses) within a program. FBFL tools leverage MaxSAT and the theory of *Model-Based Diagnosis* (MBD) (Reiter 1987; Jose and Majumdar 2011b; Marques-Silva et al. 2015; Ignatiev et al. 2019; Orvalho, Janota, and Manquinho 2024b). Moreover, these FBFL tools enumerate all diagnoses of a MaxSAT formula corresponding to bug locations.

Program Sketch. A *program sketch* is a partially incomplete program where all buggy statements are replaced by placeholders, identified as "`@ HOLES @`". These placeholders indicate parts of the program that need to be synthesized to ensure the program complies with a given specification (e.g., a test suite). Listing 3 shows a program sketch.

Abstract Syntax Tree (AST). An AST is a syntax tree in which each node represents an operation, and the node's children represent the arguments of the operation for a given programming language described by a Context-Free Grammar. An AST depicts a program's grammatical structure (Aho, Sethi, and Ullman 1986).

Counterexample Guided Automated Repair

Our approach combines the strengths of both Formal Methods (FM) and LLMs to enhance Automated Program Repair (APR). Firstly, we employ MaxSAT-based fault localization techniques to rigorously identify the minimal set of buggy parts of a program (Ignatiev et al. 2019; Orvalho, Janota, and Manquinho 2024b). Afterwards, we leverage LLMs to quickly synthesize the missing parts in the program sketch. Finally, we use a counterexample from the test suite to guide LLMs in generating patches that make the synthesized program compliant with the entire test suite, thus completing the repair. The rationale of our approach follows a Counterexample Guided Inductive Synthesis (CEGIS) (Solar-Lezama et al. 2006) loop to iteratively refine the program. Figure 1 provides an overview of our APR approach. The

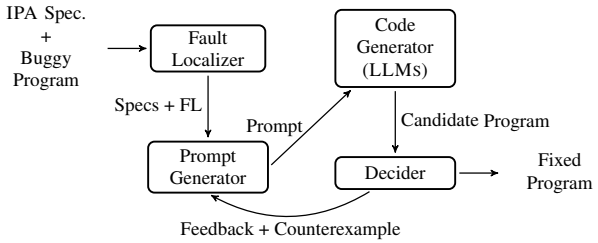

Figure 1: Counterexample Guided Automated Repair.

input is a buggy program and the specifications for an introductory programming assignment (IPA), including its description, a test suite, and the lecturer's reference solution. We start by using MaxSAT-based fault localization techniques to identify the program's minimal set of faulty statements. Next, the prompt generator builds a prompt based on the specifications of the IPA and a bug-free program sketch reflecting the localized faults, then feeds this information to the LLM. The LLM generates a program based on the provided prompt. After each iteration, the Decider module evaluates the synthesised program against a test suite. If the program is incorrect, a counterexample chosen from the test suite is sent to the prompt generator, which then feeds this counterexample to the LLM to prompt a revised synthesis.

Prompts. The prompts fed to LLMs can contain various types of information related to the IPA. The typical information available in every programming course includes the description of the IPA, the test suite to check the students' submissions corresponding to the IPA's specifications, and the lecturer's reference implementation. The syntax used in our prompts is similar to that in other works on LLM-driven program repair (Joshi et al. 2023). We have evaluated several types of prompts. Basic prompts are the simplest prompts that can be fed to an LLM without additional computation, including all the programming assignment's basic information. An example of such a prompt is shown below:

````
Fix all semantic bugs in the buggy program
below. Modify the code as little as possible.

### Problem Description ###
Write a program that determines and prints
the largest of three given integers.

### Test Suite
#input: 6 2 1
#output: 6
// The other input-output tests

# Reference Implementation (Do not copy
this program) <c> #
```c
  // Reference Implementation
```

### Buggy Program <c> ###
```c
  // Buggy program from Listing 1
```

### Fixed Program <c> ###
```c
````

In order to incorporate information about the faults localized in the program using MaxSAT-based fault localization, we utilized two different types of prompts: (1) `FIXME` annotations and (2) program sketches. `FIXME` annotated prompts are prompts where each buggy line identified by the fault localization tool is marked with a `/* FIXME */` comment. These prompts are quite similar to the basic prompt described previously, with the primary differences being the annotations in the buggy program and the first command given to the LLMs, which is modified as follows:

```
Fix all buggy lines with '/* FIXME */'
comments in the buggy program below.
```

In the second type of prompt, to address program repair as a string completion problem, we evaluated the use of prompts where the buggy program is replaced by an incomplete program (program sketch), with each line identified as buggy by our fault localization module replaced by a *hole*. The command given to the LLMs is now to complete the incomplete program. Consequently, the sections 'Buggy Program' and 'Fixed Program' are replaced by 'Incomplete Program' and 'Complete Program', respectively, as follows:

```
Complete all the '@ HOLES N @' in the
incomplete program below.
// ...
### Incomplete Program <c> ###
// ...
### Complete Program <c> ###
```

Feedback. If the candidate program generated by the LLM is not compliant with the test suite, this feedback is provided to the LLM in a new message through iterative querying. This new prompt indicates that the LLM's previous suggestion to fix the buggy program was incorrect and provides a counterexample (i.e., an IO test) where the suggested fixed program produces an incorrect output. Hence, we provide the LLM with a feedback prompt similar to:

````
### Feedback ###
Your previous suggestion was incorrect!
Try again. Code only. Provide no explanation.
### Counterexample  ###
#input: 6 2 1
#output: 6

### Fixed Program <c> ###
```c
````

Experimental Results

The goal of our evaluation is to answer the following research questions: **RQ1.** How effective are state-of-the-art (SOTA) LLMs in repairing introductory programming assignments (IPAs) compared to different SOTA semantic repair approaches? **RQ2.** How do different prompt configurations impact the performance of LLMs? **RQ3.** How does FM-based fault localization impact LLM-driven APR? **RQ4.** How helpful is it to provide a reference implementation for the same IPA to the LLMs? **RQ5.** What is the performance impact of using a Counterexample Guided approach in LLM-driven APR?

Experimental Setup. All LLMs were run using NVIDIA RTX A4000 graphics cards with 16GB of memory on an Intel(R) Xeon(R) Silver 4130 CPU @ 2.10GHz with 48 CPUs and 128GB RAM. All the experiments related to the program repair tasks were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 10GB and a timeout of 90 seconds.

Evaluation Benchmark. We evaluated our work using C-PACK-IPAS (Orvalho, Janota, and Manquinho 2024a), which consists of 1431 semantically incorrect student C programs that compile successfully but fail at least one test.

Large Language Models (LLMs). In our experiments, we used only open-access LLMs available on Hugging Face (HuggingFace 2024) with approximately 7 billion parameters for three primary reasons. Firstly, closed-access models like Chat-GPT are cost-prohibitive and raise concerns over student data privacy. Secondly, models with a very large number of parameters (e.g., 70B) need significant computational resources, such as GPUs with higher RAM capacities, and take longer to generate responses, which is unsuitable for a classroom setting. Thirdly, we used these off-the-shelf LLMs to evaluate the publicly available versions without fine-tuning them. This approach ensures that the LLMs used in this paper are available to anyone without investing time and resources into fine-tuning these models. Thus, we evaluated six different LLMs for this study through iterative querying. Three of these models are LLMCs, i.e., LLMs fine-tuned for coding tasks: IBM's GRANITE (Mishra 2024), Google's CODEGEMMA (Zhao 2024) and Meta's CODELLAMA (Rozière 2023). The other three models are general-purpose LLMs not specifically tailored for coding tasks: Google's GEMMA (Mesnard 2024), Meta's LLAMA3 (latest version of the LLAMA family (Touvron 2023)) and Microsoft's PHI3 (Abdin 2024).

We selected specific variants of each model to optimize their performance for our program repair tasks. For Meta's LLAMA3, we utilized the 8B-parameter instruction-tuned variant. This model is designed to follow instructions more accurately, making it suitable for a range of tasks, including program repair. For CODELLAMA, we used the 7B-parameter instruct-tuned version, which is specifically designed for general code synthesis and understanding, making it highly effective for coding tasks. We employed GRANITE model with 8B-parameters, fine-tuned to respond to coding-related instructions. For PHI3, we opted for the mini version, which has 3.8B-parameters and a context length of 128K. This smaller model is efficient yet capable of handling extensive context, making it practical for educational settings. For GEMMA, we used the 7B-parameter instruction-tuned version, optimized to follow detailed instructions. Lastly, for CODEGEMMA, we selected the 7B-parameter instruction-tuned variant, designed specifically for code chat and instruction, enhancing its capability to handle programming-related queries and tasks. To fit all LLMs into 16GB GPUs, we used model quantization of 4bit. Moreover, all LLMs were run using Hugging Face's Pipeline architecture. By using these different LLMs, we aimed to balance computational efficiency with the ability to effectively generate and refine code, facilitating a practical APR approach in an educational environment.

Fault Localization (FL). We used CFAULTS (Orvalho, Janota, and Manquinho 2024b) which is a formula-based FL tool, that pinpoints bug locations within the programs. It aggregates all failing test cases into a unified MaxSAT formula.

Evaluation

To assess the effectiveness of the program fixes generated by the LLMs under different prompt configurations, we used two key metrics: the number of programs successfully repaired and the quality of the repairs. For assessing the patch quality, we use the *Tree Edit Distance* (TED) (Tai 1979; Zhang and Shasha 1989) to compute the distance between the student's buggy program and the fixed program returned by the LLMs. TED computes the structural differences between two Abstract Syntax Trees (ASTs) by calculating the minimum number of edit operations (i.e., insertions, deletions, and substitutions) needed to transform one AST into another. Based on this metric for measuring program distances, we computed the *distance score*, defined by Equation 1. This score aims to identify and penalize LLMs that replace the buggy program with the reference solution rather than fixing it. The distance score is zero when the TED of the original buggy program (T_o) to the program suggested by the LLM (T_f) is the same as the TED of the reference solution (T_r) to T_o. Otherwise, it penalizes larger fixes than necessary to align the program with the correct solution.

$$ds(T_f, T_o, T_r) = max\Big(0, 1 - \frac{\text{TED}(T_f, T_o)}{\text{TED}(T_r, T_o)}\Big) \quad (1)$$

Baseline. We used two state-of-the-art traditional semantic program repair tools for IPAs as baselines: VERIFIX (Ahmed et al. 2022) and CLARA (Gulwani, Radicek, and Zuleger 2018). VERIFIX employs MaxSMT to align a buggy program with a reference implementation provided by the lecturer, while CLARA clusters multiple correct implementations and selects the one that produces the smallest fix when aligned with the buggy program. Both tools require an exact match between the control flow graphs (e.g., branches, loops) and a bijective relationship between the variables; otherwise, they return a structural mismatch error. VERIFIX was provided with each buggy program, the reference implementation, and a test suite. CLARA was given all correct programs from different academic years to generate clusters for each IPA. Within a 90-second time limit, CLARA repairs 495 programs (34.6%), times out without producing a repair on 154 programs (10.8%), and fails to repair 738 programs (54.7%). In comparison, VERIFIX repairs 91 programs (6.3%), reaches the time limit on 0.6%, and fails to repair 1338 programs (93.5%). The main reason for these failures is that both tools rely on structure mismatch errors.

Table 1 presents the number of programs successfully repaired by each LLM under various configurations. The row labeled `Portfolio` represents the best possible outcomes by selecting the optimal configuration for each program across all LLMs. Meanwhile, `Portfolio` column

Configurations without access to Reference Implementations							
LLMs	**De-TS**	**De-TS-CE**	**FIXME_De-TS**	**FIXME_De-TS-CE**	**Sk_De-TS**	**Sk_De-TS-CE**	**Portfolio (All Configurations)**
CodeGemma	597 (41.7%)	606 (42.3%)	592 (41.4%)	601 (42.0%)	682 (47.7%)	**688 (48.1%)**	823 (57.5%)
CodeLlama	492 (34.4%)	500 (34.9%)	481 (33.6%)	463 (32.4%)	**573 (40.0%)**	561 (39.2%)	712 (49.8%)
Gemma	496 (34.7%)	492 (34.4%)	446 (31.2%)	444 (31.0%)	532 (37.2%)	**534 (37.3%)**	670 (46.8%)
Granite	626 (43.7%)	624 (43.6%)	566 (39.6%)	583 (40.7%)	**691 (48.3%)**	681 (47.6%)	846 (59.1%)
Llama3	564 (39.4%)	590 (41.2%)	535 (37.4%)	557 (38.9%)	578 (40.4%)	**591 (41.3%)**	851 (59.5%)
Phi3	494 (34.5%)	489 (34.2%)	460 (32.1%)	474 (33.1%)	**547 (38.2%)**	535 (37.4%)	621 (43.4%)
Portfolio (All LLMs)	842 (58.8%)	846 (59.1%)	796 (55.6%)	820 (57.3%)	900 (62.9%)	**907 (63.4%)**	1013 (70.8%)

Configurations with access to Reference Implementations							
LLMs	**De-TS-CE-CPA**	**De-TS-CE-RI**	**FIXME_De-TS-CE-CPA**	**FIXME_De-TS-CE-RI**	**Sk_De-TS-CE-CPA**	**Sk_De-TS-CE-RI**	**Portfolio (All Configurations)**
CodeGemma	578 (40.4%)	576 (40.3%)	637 (44.5%)	638 (44.6%)	725 (50.7%)	**739 (51.6%)**	916 (64.0%)
CodeLlama	528 (36.9%)	525 (36.7%)	565 (39.5%)	609 (42.6%)	633 (44.2%)	**675 (47.2%)**	893 (62.4%)
Gemma	595 (41.6%)	607 (42.4%)	563 (39.3%)	616 (43.0%)	664 (46.4%)	**732 (51.2%)**	951 (66.5%)
Granite	773 (54.0%)	828 (57.9%)	794 (55.5%)	857 (59.9%)	838 (58.6%)	**876 (61.2%)**	1132 (79.1%)
Llama3	685 (47.9%)	691 (48.3%)	657 (45.9%)	681 (47.6%)	725 (50.7%)	**730 (51.0%)**	1016 (71.0%)
Phi3	552 (38.6%)	444 (31.0%)	545 (38.1%)	492 (34.4%)	639 (44.7%)	**647 (45.2%)**	899 (62.8%)
Portfolio (All LLMs)	1033 (72.2%)	1046 (73.1%)	1011 (70.6%)	1056 (73.8%)	1050 (73.4%)	**1077 (75.3%)**	1190 (83.2%)

Table 1: The number of programs fixed by each LLM under various configurations. Row *Portfolio (All* LLMs*)*, shows the best results across all LLMs for each configuration. Column *Portfolio (All Configurations)* shows the best results for each LLM across all configurations. Mapping abbreviations to configuration names: **De** - IPA *Description*, **TS** - *Test Suite*, **CE** - *Counterexample*, **RI** - *Reference Implementation*, **CPA** - *Closest Program using* ASTs, **FIXME** - *FIXME Annotations*, **SK** - *Sketches*.

Metric: **sum(Distance Score)**								
	Configurations							
LLMs	**De-TS**	**De-TS-CE**	**De-TS-CE-CPA**	**De-TS-CE-RI**	**Sk_De-TS**	**Sk_De-TS-CE**	**Sk_De-TS-CE-CPA**	**Sk_De-TS-CE-RI**
CodeGemma	471.0	486.4	429.7	440.4	524.4	**529.5**	249.8	497.3
CodeLlama	437.5	438.8	409.5	404.8	**477.9**	464.5	251.3	459.0
Gemma	306.5	296.9	**370.8**	231.0	338.8	340.3	156.4	316.2
Granite	512.8	506.3	453.4	292.1	**539.8**	533.6	172.3	334.5
Llama3	367.9	368.0	414.8	381.9	379.8	384.5	172.7	**423.0**
Phi3	291.9	292.6	287.6	148.1	**326.5**	321.4	98.2	253.4

Table 2: The cumulative distance scores for each program successfully repaired by each LLM across various configurations.

highlights the best results achieved by a particular LLM across all tested configurations. The configurations yielding the highest success rates for the six evaluated LLMs involve incorporating a reference implementation of the IPA into the prompt. However, rather than genuinely fixing the buggy program, the LLMs often replace it with the reference implementation. For instance, GRANITE repairs 876 programs using a configuration that includes bug-free program sketches (Sk), an IPA description, counterexamples, a test suite, and the reference implementation (Sk_De-TS-CE-RI). Notably, 442 of these repaired programs exhibit a TED value of zero between the reference implementation and the fixed program, indicating that GRANITE is replicating the reference implementation. To address this, we separately analyzed configurations that include and exclude access to a reference implementation. When no reference implementation is provided (top of Table 1), GRANITE still leads among the LLMs, fixing up to 59.1% of the programs across all configurations and 48.3% when using sketches (SK), the IPA description, and a test suite (SK_De-TS). CODEGEMMA also performs well, achieving up to 57.5% success in a portfolio approach and showing particular strength in configurations involving sketches (SK). For instance, CODEGEMMA can repair 48.1% of the evaluation benchmark using bug-free sketches, IPA description, test suite, and counterexample (SK_De-TS-CE). Configurations incorporating

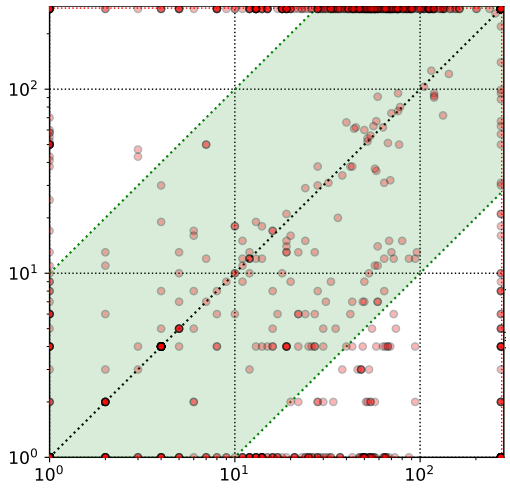

Figure 2: Comparison of tree edit distances (TED) for GRANITE's repairs when using (x-axis) versus not using (y-axis) correct implementations with configuration Sk_De-TS-CE.

sketches (SK) and FIXME annotations generally yield better results. Including counterexamples (CE), IPA descriptions, and test suites (De-TS) further boosts the success rate across different LLMs. The portfolio approach, which com-

bines the strengths of all LLMs and configurations without using reference implementation, achieves the highest overall success rate, fixing 70.8% of the programs. This demonstrates that leveraging multiple LLMs together can significantly enhance repair success.

Furthermore, we provide the results of LLMs with a reference implementation (bottom of Table 1). The reference implementation can be either the lecturer's implementation for the same IPA or the closest correct program based on the programs' Abstract Syntax Trees (ASTs) from a previously submitted student program, determined by Tree Edit Distance (TED) values (Orvalho, Janota, and Manquinho 2022a). The intent was to allow the model to reuse correct code snippets to generate repairs. Results show that including a reference implementation allows for better repair results. However, as mentioned earlier, the LLMs often simply copy the provided reference implementation. Table 2 presents the sum of the distance scores (see Eq. 1) for the top-performing LLMs from Table 1 across different configurations. This summation aims to penalize LLMs that either copy the provided reference implementation or generate unnecessarily large repairs. For example, GRANITE using configuration Sk_De-TS-CE-RI can repair 876 programs but yields a total distance score of 334.5, whereas using the same configuration without a correct implementation repairs 681 programs resulting in a higher distance score of 533.6.

Figure 2 shows a scatter plot that compares the tree edit distance (TED) of the buggy program to the program fixed by GRANITE with and without a reference implementation, using configuration Sk_De-TS-CE. Each point represents a faulty program, where the x-value (resp. y-value) represents the TED cost of GRANITE' with access to a reference implementation (resp. without it). Points below the diagonal indicate that fixing a program with access to a correct implementation incurs a higher TED cost than fixing it without access. This suggests that while access to a reference implementation enables GRANITE and other LLMs to repair more programs, it often results in larger changes to the student's program than when no correct implementation is given.

Discussion. To answer our research questions: For RQ1, all six LLMs using different prompt configurations repair more programs than traditional repair tools. For RQ2, prompt configurations with FL-based Sketches, IPA description and test suite yield the most successful repair outcomes. Moreover, for RQ3, it is clear that incorporating FL-based Sketches (or even FIXME annotations) allows the LLMs to repair more programs than only providing the buggy program. For RQ4, including a reference implementation allows for more repaired programs but with potentially less efficient fixes. Finally, for RQ5, employing a Counterexample guided approach significantly improves the accuracy of LLM-driven APR across various configurations. Counterexamples help in the repair process of certain LLMs, such as CODEGEMMA and LLAMA3, across all prompt configurations. For other LLMs, counterexamples are beneficial but only in specific configurations. This difference may be due to variations in the training data used for each LLM. Moreover, we analyzed the effectiveness of LLMs in repairing programs that CLARA fails to address due to control-flow issues, finding that GRANITE with Sk_De-TS performed best, fixing 37.0% of cases, while CODEGEMMA with Sk_De-TS-CE followed with 34.5%. GRANITE also demonstrated superior performance in cases with higher average cyclomatic complexity, whereas CODEGEMMA was most effective for simpler programs.

Related Work

Several constraint-based program repair techniques have been proposed to check if a student's program is semantically correct: clustering-based (Gulwani, Radicek, and Zuleger 2018), implementation-driven (Wang, Singh, and Su 2018; Ahmed et al. 2022; Hu et al. 2019; Liu et al. 2019), and semantic code search (Afzal et al. 2019). *Clustering-based repair* tools (Gulwani, Radicek, and Zuleger 2018) receive an incorrect program, a test suite, and a set of correct student submissions for the same IPA. *Implementation-driven repair* tools use one reference implementation to repair a given incorrect submission (Ahmed et al. 2022).

Large Language Models (LLMs) trained on code (LLMCs) have demonstrated significant effectiveness in generating program fixes (Joshi et al. 2023; Xia, Ding, and Zhang 2023; Wei, Xia, and Zhang 2023; Fan et al. 2023; Xia, Wei, and Zhang 2023; Brancas, Manquinho, and Martins 2024). For instance, RING (Joshi et al. 2023) is a multilingual repair engine powered by an LLMC that uses fault localization (FL) information from error messages and leverages the few-shot capabilities of LLMCs for code transformation. In the context of Automated Program Repair (APR) for programming education, several works have explored the use of LLMs for coding tasks (Zhang et al. 2024; Phung et al. 2023; Liffiton et al. 2023). PyDex (Zhang et al. 2024), for example, employs iterative querying with CODEX, an LLMC version of ChatGPT, using test-based few-shot selection and structure-based program chunking to repair syntax and semantic errors in Python assignments. Similarly, CODEHELP (Liffiton et al. 2023) utilizes OpenAI's LLMs to provide textual feedback to students on their assignments. However, to the best of our knowledge, no existing work has explored the use of LLMs guided by formula-based FL.

Conclusion

Large Language Models (LLMs) excel at completing strings, while MaxSAT-based fault localization (FL) excels at identifying buggy parts of a program. We proposed a novel approach combining MaxSAT-based FL and LLMs via zero-shot learning to enhance Automated Program Repair (APR) for introductory programming assignments (IPAs). Experiments show that our bug-free program sketches, significantly improves the repair capabilities of all six evaluated LLMs, enabling them to repair more programs and produce smaller patches compared to other configurations and state-of-the-art symbolic program repair tools. Therefore, this interaction between Formal Methods and LLMs yields more accurate and efficient program fixes, enhancing feedback mechanisms in programming education.

Acknowledgments

PO acknowledges support from the EU's Horizon 2020 research and innovation programme under ELISE Grant Agreement No 951847 and the ERC AdG FUN2MODEL (Grant agreement No. 834115). This work was partially supported by Portuguese national funds through FCT, under projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/-2020), PTDC/CCI-COM/2156/2021 (DOI: 10.54499/-PTDC/CCI-COM/2156/2021) and 2023.14280.PEX (DOI: 10.54499/2023.14280.PEX) and grant SFRH/BD/07724/-2020 (DOI: 10.54499/2020.07724.BD). This work was also supported by the MEYS within the program ERC CZ under the project POSTMAN no. LL1902 and co-funded by the EU under the project *ROBOPROX* (reg. no. CZ.02.01.01/00/22_008/0004590).

References

Abate, A.; David, C.; Kesseli, P.; Kroening, D.; and Polgreen, E. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *CAV 2018*, volume 10981 of *LNCS*, 270–288. Springer.

Abdin, e.-O. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. *CoRR*, abs/2404.14219.

Afzal, A.; Motwani, M.; Stolee, K. T.; Brun, Y.; and Goues, C. L. 2019. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Trans. Software Eng.*, 47(10): 2162–2181.

Ahmed, U. Z.; Fan, Z.; Yi, J.; Al-Bataineh, O. I.; and Roychoudhury, A. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Trans. Softw. Eng. Methodol.*

Aho, A. V.; Sethi, R.; and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley.

Bhatia, S.; Kohli, P.; and Singh, R. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *ICSE 2018*, 60–70. ACM.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Brancas, R.; Manquinho, V.; and Martins, R. 2024. Combining Logic with Large Language Models for Automatic Debugging and Repair of ASP Programs. *CoRR*, abs/2410.20962.

Contractor, M. R.; and Rivero, C. R. 2022. Improving Program Matching to Automatically Repair Introductory Programs. In Crossley, S.; and Popescu, E., eds., *Intelligent Tutoring Systems*, 323–335. Cham: Springer International Publishing.

Fan, Z.; Gao, X.; Mirchev, M.; Roychoudhury, A.; and Tan, S. H. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, 1469–1481. IEEE.

Gulwani, S.; Radicek, I.; and Zuleger, F. 2018. Automated clustering and program repair for introductory programming assignments. In *PLDI 2018*, 465–480. ACM.

Gupta, R.; Kanade, A.; and Shevade, S. K. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, 930–937. AAAI Press.

Gupta, R.; Pal, S.; Kanade, A.; and Shevade, S. K. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In Singh, S. P.; and Markovitch, S., eds., *AAAI 2017*, 1345–1351. AAAI Press.

Hu, Y.; Ahmed, U. Z.; Mechtaev, S.; Leong, B.; and Roychoudhury, A. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 388–398. IEEE.

HuggingFace. 2024. . https://huggingface.co. [Online; accessed 1-July-2024].

Ignatiev, A.; Morgado, A.; Weissenbacher, G.; and Marques-Silva, J. 2019. Model-Based Diagnosis with Multiple Observations. In Kraus, S., ed., *IJCAI 2019*, 1108–1115. ijcai.org.

Jha, S.; Gulwani, S.; Seshia, S. A.; and Tiwari, A. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 215–224.

Jin, M.; Shahriar, S.; Tufano, M.; Shi, X.; Lu, S.; Sundaresan, N.; and Svyatkovskiy, A. 2023. InferFix: End-to-End Program Repair with LLMs. In *ESEC/FSE 2023*, 1646–1656. ACM.

Jose, M.; and Majumdar, R. 2011a. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV 2011*, volume 6806 of *LNCS*, 504–509. Springer.

Jose, M.; and Majumdar, R. 2011b. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, 437–446. ACM.

Joshi, H.; Sánchez, J. P. C.; Gulwani, S.; Le, V.; Verbruggen, G.; and Radicek, I. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *AAAI 2023*, 5131–5140. AAAI Press.

Liffiton, M. H.; Sheese, B. E.; Savelka, J.; and Denny, P. 2023. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. In *Koli Calling International Conference on Computing Education Research*, 8:1–8:11. ACM.

Liu, X.; Wang, S.; Wang, P.; and Wu, D. 2019. Automatic grading of programming assignments: an approach based on formal semantics. In Beecham, S.; and Damian, D. E., eds., *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2019*, 126–137. IEEE / ACM.

Marques-Silva, J.; Janota, M.; Ignatiev, A.; and Morgado, A. 2015. Efficient Model Based Diagnosis with Maximum Satisfiability. In *IJCAI 2015*, 1966–1972. AAAI Press.

Mesbah, A.; Rice, A.; Johnston, E.; Glorioso, N.; and Aftandilian, E. 2019. DeepDelta: learning to repair compilation errors. In *ESEC/SIGSOFT FSE 2019*, 925–936. ACM.

Mesnard, e.-a. 2024. Gemma: Open Models Based on Gemini Research and Technology. *CoRR*, abs/2403.08295.

Mishra, e.-a. 2024. Granite Code Models: A Family of Open Foundation Models for Code Intelligence. *CoRR*, abs/2405.04324.

Orvalho, P.; Janota, M.; and Manquinho, V. 2022a. InvAASTCluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. *CoRR*, abs/2206.14175.

Orvalho, P.; Janota, M.; and Manquinho, V. 2024a. C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. In *2024 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 14–21. .: ACM.

Orvalho, P.; Janota, M.; and Manquinho, V. 2024b. CFaults: Model-Based Diagnosis for Fault Localization in C Programs with Multiple Test Cases. In *Formal Methods - 26th International Symposium, FM 2024*, volume 14933 of *Lecture Notes in Computer Science*, 463–481. ISBN 978-3-031-71162-6.

Orvalho, P.; Janota, M.; and Manquinho, V. 2024c. GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. In *Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, SIGCSE Virtual 2024, Virtual Event, NC, USA, December 5-8, 2024*. ACM.

Orvalho, P.; Janota, M.; and Manquinho, V. M. 2022b. MultIPAs: Applying Program Transformations To Introductory Programming Assignments For Data Augmentation. In *ESEC/FSE 2022*, 1657–1661. ACM.

Orvalho, P.; Manquinho, V.; and Martins, R. 2023. UpMax: User Partitioning for MaxSAT. In *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023*, volume 271 of *LIPIcs*, 19:1–19:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Orvalho, P.; Piepenbrock, J.; Janota, M.; and Manquinho, V. M. 2023. Graph Neural Networks for Mapping Variables Between Programs. In *ECAI 2023 - 26th European Conference on Artificial Intelligence*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, 1811–1818. Poland: IOS Press.

Orvalho, P.; Terra-Neves, M.; Ventura, M.; Martins, R.; and Manquinho, V. M. 2019. Encodings for Enumeration-Based Program Synthesis. In *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, 583–599.

Phung, T.; Cambronero, J.; Gulwani, S.; Kohn, T.; Majumdar, R.; Singla, A.; and Soares, G. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. In *International Conference on Educational Data Mining, EDM*. International Educational Data Mining Society.

Pu, Y.; Narasimhan, K.; Solar-Lezama, A.; and Barzilay, R. 2016. sk_p: a neural program corrector for MOOCs. In Visser, E., ed., *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016*, 39–40. ACM.

Reiter, R. 1987. A Theory of Diagnosis from First Principles. *Artif. Intell.*, 32(1): 57–95.

Rolim, R.; Soares, G.; D'Antoni, L.; Polozov, O.; Gulwani, S.; Gheyi, R.; Suzuki, R.; and Hartmann, B. 2017. Learning syntactic program transformations from examples. In Uchitel, S.; Orso, A.; and Robillard, M. P., eds., *ICSE 2017*, 404–415. IEEE / ACM.

Rozière, e.-a. 2023. Code Llama: Open Foundation Models for Code. *CoRR*, abs/2308.12950.

Solar-Lezama, A.; Rabbah, R. M.; Bodík, R.; and Ebcioglu, K. 2005. Programming by sketching for bit-streaming programs. In *PLDI*, 281–294. ACM.

Solar-Lezama, A.; Tancau, L.; Bodík, R.; Seshia, S. A.; and Saraswat, V. A. 2006. Combinatorial sketching for finite programs. In *ASPLOS*, 404–415.

Tai, K. 1979. The Tree-to-Tree Correction Problem. *J. ACM*, 26(3): 422–433.

Touvron, e.-a. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR*, abs/2302.13971.

Wang, K.; Singh, R.; and Su, Z. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *PLDI 2018*, 481–495. ACM.

Wei, Y.; Xia, C. S.; and Zhang, L. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *ESEC/FSE*, 172–184. ACM.

Xia, C. S.; Ding, Y.; and Zhang, L. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, 522–534. IEEE.

Xia, C. S.; Wei, Y.; and Zhang, L. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *ICSE*, 1482–1494. IEEE.

Xia, C. S.; and Zhang, L. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *ESEC/FSE*, 959–971. ACM.

Yasunaga, M.; and Liang, P. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In *ICML 2020*, volume 119, 10799–10808. PMLR.

Zhang, J.; Cambronero, J. P.; Gulwani, S.; Le, V.; Piskac, R.; Soares, G.; and Verbruggen, G. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. ., 8(OOPSLA): 1100–1124.

Zhang, K.; and Shasha, D. E. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18(6): 1245–1262.

Zhao, e.-a. 2024. CodeGemma: Open Code Models Based on Gemma. *CoRR*, abs/2406.11409.